Microsoft® Windows
Device Development Kit

development tools for providing Microsoft® Windows device support

Virtual Device Adaptation Guide

VERSION 3.0

for the MS-DOS® Operating System

Microsoft Corporation

Printed and bound in the United States of America.

Document No. xxxx
Part No. yyyy
10 9 8 7 6 5 4 3 2 1

Table of Contents

Virtual Device Adaptation Guide

PART 3 Writing Virtual Devices

PART 4 Virtual Device Services

Appendixes

Introduction to Virtual Devices

This document explains how to modify existing device drivers or create new virtual devices that will work with Microsoft Windows 3.0 when running in 386 enhanced mode.

This introduction provides some background information that you should review before using this documentation. The topics are presented in the following order:

- What you should know before you start
- Organization of this document
- Notational conventions

What You Should Know Before You Start

To program virtual devices for Windows when running in 386 enhanced mode, you should be familiar with Part 1, "Writing Windows Device Drivers," in the *Microsoft Windows Device Driver Adaptation Guide* and the following topics. Suggested reference materials are shown by topic:

Topics	Reference
MS-DOS	Duncan, Ray. *Advanced MS-DOS*. Microsoft Press, P.O. Box 97017, Redmond WA. 98073-9717. ISBN Number: 0-914845-77-2
	MS-DOS Encyclopedia. Microsoft Press, P.O. Box 97017, Redmond WA. 98073-9717. ISBN Number: 1-5565-174-8
Microsoft Windows 3.0, (especially the Memory Management topics)	*Microsoft Windows 3.0 Software Development Kit*, "Programming Topics"
Assembly-language programming for the Intel 80386 microprocessor	Ahern-Wahlstrom. *Intel 80386 Programmer's Reference*. Intel Literature Sales, P.O. Box 58130, Santa Clara, CA. 95052-8130. Order Number: 230985-8130

Organization of This Document

This document is divided into the following parts and chapters:

Part 3, "Writing Virtual Devices," describes the requirements of a virtual device, and the environment of Windows when running in 386 enhanced mode. Part 3 contains the following chapters:

Chapter 16, "Overview of Windows in 386 Enhanced Mode," which provides the conceptual foundation of the Windows virtual machine environment.

Chapter 17, "Virtual Device Programming Topics," which provides a more in-depth look at various programming topics.

Chapter 18, "The VDD and Grabber DLL," which describes the development of a Virtual Display Driver (VDD) and the dynamic-link library (DLL) needed to support a video adapter.

Part 4, "Virtual Device Services," provides detailed descriptions of all the available services. It consists of the following 23 chapters:

- Chapter 19, "Memory Management Services"
- Chapter 20, "I/O Services and Macros"
- Chapter 21, "VM Interrupt and Call Services"
- Chapter 22, "Nested Execution Services"
- Chapter 23, "Break Point and Callback Services"
- Chapter 24, "Primary Scheduler Services"
- Chapter 25, "Time-Slice Scheduler Services"
- Chapter 26, "Event Services"
- Chapter 27, "Timing Services"
- Chapter 28, "Processor Fault and Interrupt Services"
- Chapter 29, "Information Services"
- Chapter 30, "Initialization Information Services"
- Chapter 31, "Linked List Services"
- Chapter 32, "Error Condition Services"
- Chapter 33, "Miscellaneous Services"
- Chapter 34, "Shell Services"
- Chapter 35, "Virtual Display Device (VDD) Services"
- Chapter 36, "Virtual Keyboard Device (VKD) Services"
- Chapter 37, "Virtual PIC Device (VPICD) Services"
- Chapter 38, "Virtual Sound Device (VSD) Services"

- Chapter 39, "Virtual Timer Device (VTD) Services"
- Chapter 40, "V86 Mode Memory Manager Device Services"
- Chapter 41, "Virtual DMA Device (VDMAD) Services"

Part 5, "Appendixes," provides the following supplemental reference materials:

- Appendix A, "Terms and Acronyms"
- Appendix B, "Understanding Modes"
- Appendix C, "Creating Distribution Disks for Drivers"
- Appendix D, "Windows INT 2FH API"

Notational Conventions

The following notational conventions are used throughout the DDK documentation set.

Convention	Meaning
bold	Bold is used for keywords, such as function, register, macro, and data structure field names. These names are spelled exactly as they should appear in source programs. Notice the bold in the following example: **Disable** (*lpDestDev*) Here, **Disable** is bold to indicate that it is the name of a function.
italics	Italics are used to indicate a placeholder that should be replaced by an actual argument. In the preceding example, *lpDestDev* is italic to indicate that it should be replaced by an argument.
(Parentheses)	Parentheses enclose the parameter or parameters that are to be passed to a function. In the preceding example, *lpDestDev* is the parameter.
`Monospace`	Monospace type is used for program code fragments and to illustrate the syntax of data structures.

Part 3 Writing Virtual Devices

Microsoft Windows 3.0, while running in 386 enhanced mode, allows single-threaded multitasking by creating a virtual machine environment. While this may be a new type of environment for many programmers, the advantages of freeing existing programs from the limitations of older hardware architectures should make the effort of learning it worthwhile.

To run in the enhanced Windows environment, existing device drivers will need to be modified into *virtual* devices. In Part 1, "Writing Windows Device Drivers," of the *Microsoft Windows Device Driver Adaptation Guide* the question of how long will it take to convert an existing device driver is examined.

Part 3 provides the overall concepts and functional descriptions of the environment components that are necessary to write virtual devices.

CHAPTERS

Chapter 16 Overview of Windows in 386 Enhanced Mode

When Microsoft Windows 3.0 is loaded and invoked on an appropriately configured system, it runs in an "enhanced" mode designed to capitalize on the power of the Intel 80386 microprocessor. The 386 chip, in addition to an accelerated clock, a wider data path, and an expanded command set, has a mode that supports multiple, independent memory regions. Enhanced Windows uses this microprocessor mode, the virtual 8086 mode, to build multiple, independent virtual machines, each capable of running an application program.

Enhanced Windows supports this multitasking virtual machine environment with a sophisticated set of services, many provided by the virtual devices. Virtual devices (VxDs) provide access to all the system resources, including memory management and scheduling, and to all the hardware devices. VxDs are analogous to, and often modifications of, device drivers used in other Windows modes.

By writing a VxD for a particular hardware device, the author integrates that device into the powerful enhanced Windows environment. For instance, a properly implemented virtual printer device will, by serializing access to the hardware port, enable two active applications to share a single printer.

This chapter provides a general description of the virtual machine environment and introduces the components of a virtual device. However, detailed programming instructions for the 80386 are not provided. Before proceeding, a VxD programmer should already be familiar with the topics described in the "What You Should Know Before You Start" section in the "Introduction to Virtual Devices" at the beginning of this document.

In the September, 1987, issue of the *Microsoft Systems Journal*, the article entitled "Microsoft Windows/386: Creating a Virtual Machine Environment," discusses the structure of Windows/386 version 2.x. It also contains an excellent description of the four modes of the Intel 80386 microprocessor. A portion of that discussion is included in Appendix B, "Understanding Modes," to help you understand the current version of Windows when running in 386 enhanced mode.

16.1 The Operating Environment

Windows in 386 enhanced mode has a virtual machine (VM) architecture that provides preemptive multitasking for DOS applications on the 80386 processor.

The following are its three major components, which are also graphically represented in Figure 16.1:

- Virtual machines

- Virtual Machine Manager
- Virtual devices

Windows 3.0 virtual machines (VMs) consist of a virtual 8086 (V86) mode portion and, optionally, a protected-mode (PM) portion. The first VM created is called the System VM. This is the virtual machine in which the Windows graphic user interface runs. Non-Windows applications run in VMs of their own.

The Virtual Machine Manager (VMM) functions as a multitasking operating environment. The VMM provides services that control the main memory, the CPU execution time, and the peripheral devices. It runs, along with all the VxDs, in one, flat-model, 32-bit memory segment.

The virtual devices (VxDs) either virtualize a peripheral device, provide services for the VMM and VxDs, or both. The "x" in VxD stands for an arbitrary device. In an actual device name, the "x" is replaced with the name of the virtualized device, e.g., VDD for Virtual Display Device and VDMAD for Virtual DMA Device.

Devices, such as the programmable interrupt controller and printers, are shown outside of the enhanced Windows virtualized environment.

Notice that the hardware device may consist of software, e.g., routine (BIOS) as well as hardware.

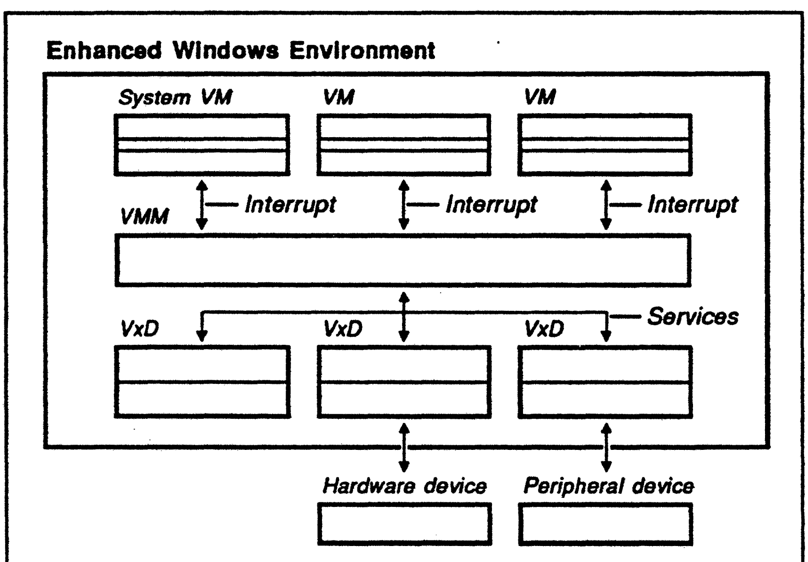


Figure 16.1 An OverAll Block Diagram INTVD_01.EPS

16.2 Virtual Machines

When Windows is running in 386 enhanced mode, it creates memory partitions that have a remarkable characteristic: programs that run within these partitions execute as though they were running on an 80386 in real mode. Each of these partitions is called a virtual 8086 machine and has its own address space, I/O port space, and interrupt vector table. Multiple virtual machines can be running simultaneously, with each under the illusion that it is in complete control of the computer.

A virtual machine (VM) is a complete description of the state of an application. Each VM includes the following:

- The memory associated with the application
- The processor registers
- The data structures associated with virtualization

Data is used by the VMM and VxDs to virtualize the hardware and to provide services. It is maintained in a data structure called the Control Block. The processor registers are maintained on the VMM stack and can be accessed via the Client Register Structure. The memory can be accessed and manipulated by means of a number of VMM memory manager services.

To optimize the use of memory and minimize the enhanced Windows environment overhead, most of MS-DOS and all the MS-DOS device drivers are not duplicated for each VM, but rather are shared (global) among the VMs.

16.2.1 The Privilege Rings of a VM

A VM can have more than one privilege ring. Code executing in one privilege ring can only have access to memory in the same privilege ring or one with a higher number (i.e., lower privilege level).

The 8086 (V86) mode portion, shown in Figure 16.2, runs in privilege ring 3. This is the code and data most typically associated with MS-DOS applications.

The second part is a protected mode (PM) portion that runs at privilege ring 1, 2, or 3. This portion can be used by applications running under enhanced Windows. In the System VM (SYS VM), this portion is used to run Windows 3.0 code.

The third part is data utilized by the VMM and the VxDs running at privilege ring 0.

The Ring 0 data has three subparts:

1. The stack, which contains the Client Register Structure (CRS). The ring 0 stack is used by the VMM and VxDs when a VM is running.

2. The control block, which contains other data (i.e., values associated with the virtualization of hardware for a VM) local to a VM.
3. Data owned by a VxD, which contains information that maintains the state, such as the state of the physical hardware, across all VMs.

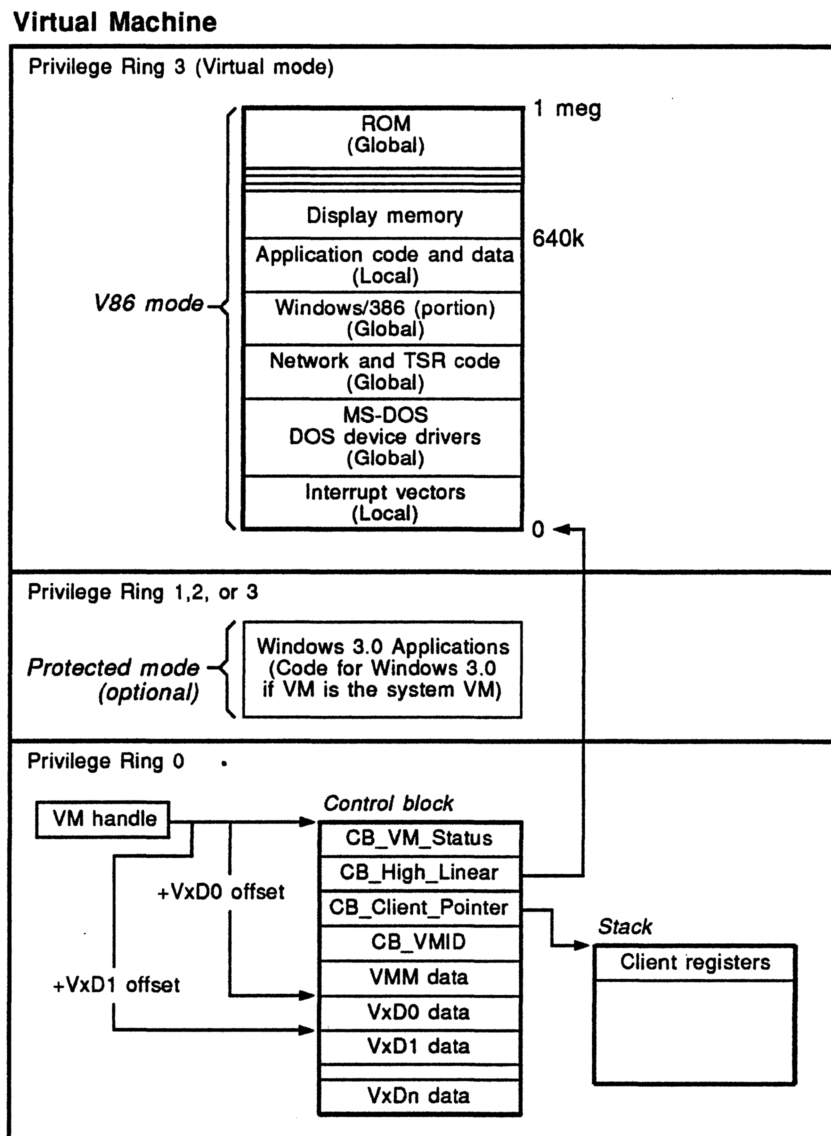


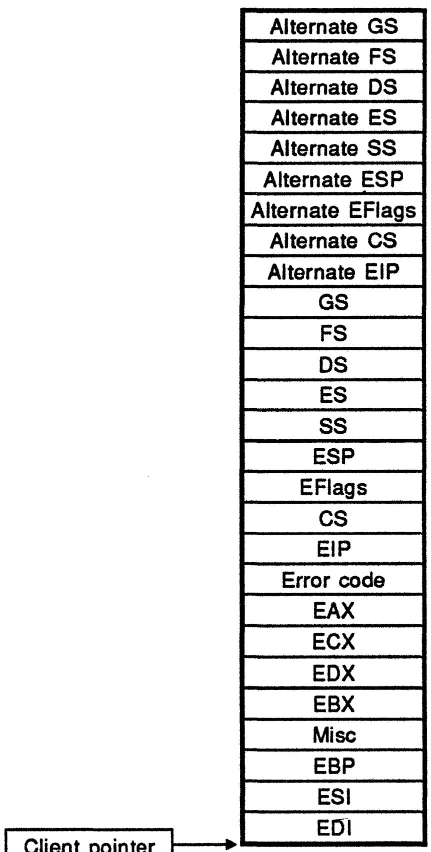
Figure 16.2 The Conceptual Detail of VMs INTVD_02.EPS

16.2.2 VM Handles

Enhanced Windows virtual devices refer to specific VMs by VM handles. By convention, VM handles are usually stored in the **EBX** register. A VM handle is actually a 32-bit linear address of the virtual machine's control block data structure.

16.2.3 The Client Register Structure

The Client Register Structure (CRS), as shown in Figure 16.3, contains the virtual machine processor state including all the virtual machine's registers and flags. When a device wants to look at or modify a virtual machine's registers, it must modify the CRS.



Note: Alternate registers hold values of mode not in use, i.e., when in V86 mode, alternate registers hold protected mode values.

Figure 16.3 The Client Register Structure INTVD_03.EPS

16.3 The Virtual Machine Manager

The Virtual Machine Manager (VMM) is a device-independent layer of code that provides a framework upon which the virtual devices build virtualizations of physical devices or provide services for each of the VMs. In this sense, the VMM lies between the VMs and the VxDs. All interaction between the software running in the VMs and the VxDs occurs via the interface provided by the VMM. The VMM also provides a set of services that allows for creating, destroying, running, synchronizing, and altering the state of the VMs. The VMM, as shown in Figure 16.4, handles all the transitions of VMs to privilege ring 0, provides scheduling services, manages memory, and provides services for such activities as trapping I/O and hooking software interrupts.

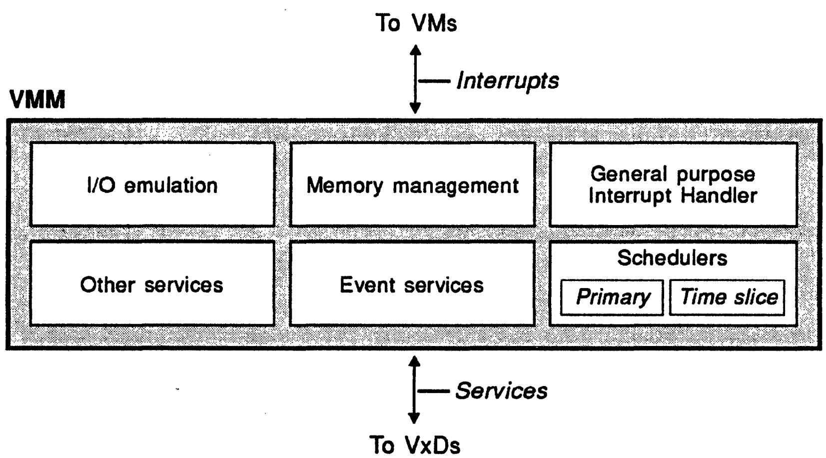


Figure 16.4 The VMM Functions INTVD_04.EPS

16.4 Virtual Devices

Enhanced Windows virtual devices (VxDs) are the interfaces between application software and the hardware. Most VxDs correspond to a hardware device, though not all do. For example, the VxDs for printers and displays simulate actual hardware interfaces, but the VxD called Shell provides access to the Windows graphic user interface. VxDs use services provided by the VMM and other VxDs.

VxDs can provide control functions, service functions, API functions, and callback procedures that are used to virtualize, synchronize, and maintain the state of the hardware for the VMs. A callback procedure is a request for notification when a specified event occurs in the normal execution of the application code.

There must be a VxD for each piece of hardware that can have a different state in each of the VMs.

16.4.1 VxD Components

Installable virtual devices have the following five, distinct parts, which are shown graphically in Figure 16.5:

1. Real mode initialization code and data, which is discarded after loading parts 2 - 5
2. Protected mode (PM) initialization code, which is discarded after initialization
3. Protected mode (PM) initialization data, which is discarded after initialization
4. PM code, which contains the Device Control Procedure, API and callback procedures, and services.
5. PM data, which contains the Device Descriptor Block, Service Table, and Global Data

16.4.2 The Device Control Procedure

The Device Control Procedure (DCP) is the dispatch point for most of the VMM interaction with the VxD. Besides the initialization of the system, there are device control calls for creating, initializing, and destroying VMs; for setting the device focus to a VM; and for indicating a change in the state of the VM.

The VMM broadcasts messages to all VxD DCPs indicating changes in the state of the system or of a VM. The DCP can then modify the device's data structure or the VM's state. The address of the DCP is specified in a special data structure called a Device Descriptor Block that all virtual devices must have. See Chapter 17, "Virtual Device Programming Topics," for details on messages passed to the DCP.

16.4.3 The Device Descriptor Block

The Device Descriptor Block (DDB) is a VxD-unique data structure containing the VxD's name, version IDs, and entry points for the three code areas: the Device Control Procedure, V86-mode API procedure, and the PM API procedure. In addition, the DDB can contain a pointer to a table of services provided by the VxD. See Chapter 17, "Virtual Device Programming Topics, " for a detailed description of a DDB.

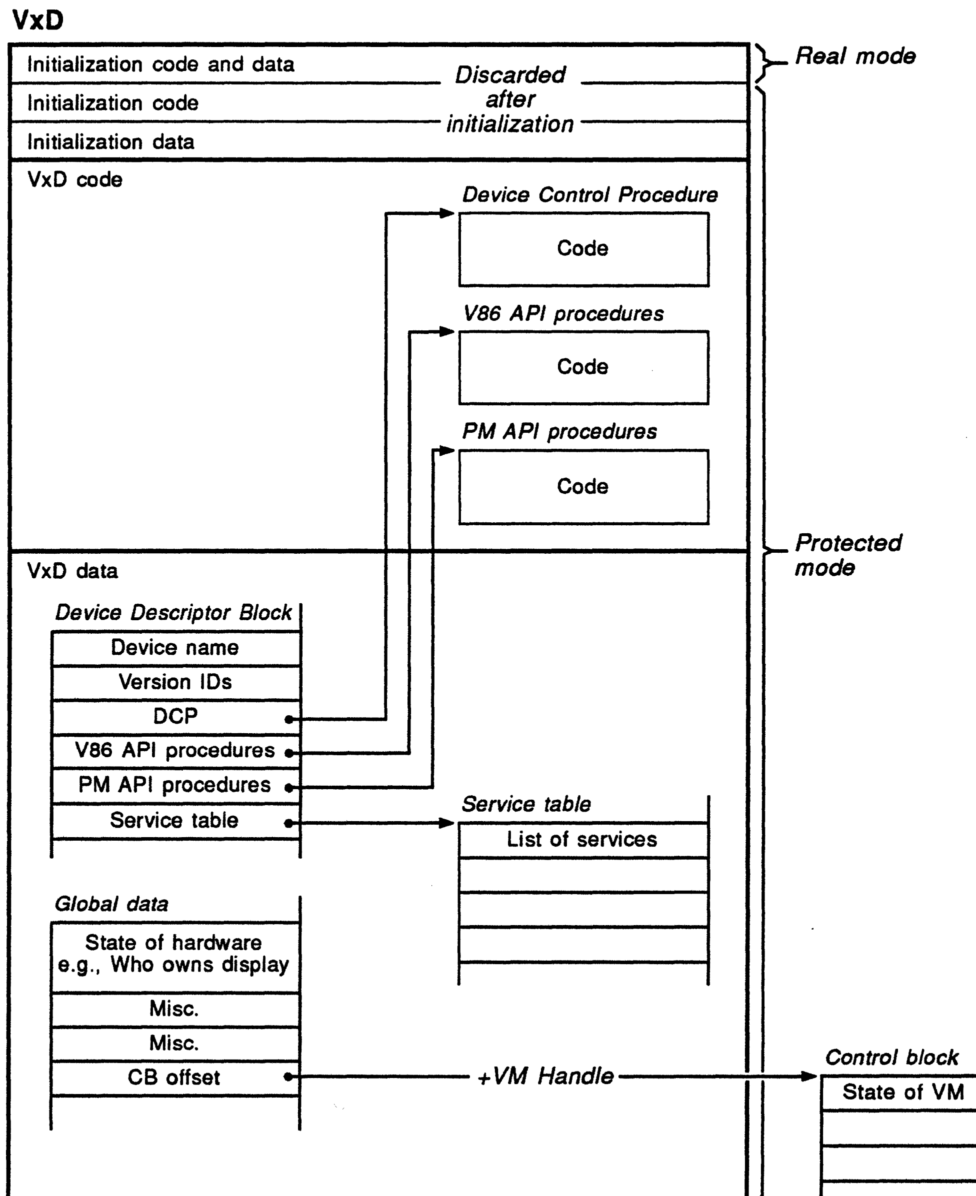


Figure 16.5 The Conceptual Detail of VxDs INTVD_05.EPS

16.5 How VxDs Work

The following sections contain general explanations of how VxDs work and provide information on the following topics:

- Scheduling
- Memory use
- Services
- Callback procedures
- I/O port traps
- Loading

16.5.1 Enhanced Windows Execution Scheduling

The following is a brief description of how events are scheduled and processed. The concepts are also graphically described in Fibure 16.6.

Events

The enhanced Windows VMM is a single-threaded, non-reentrant operating system. Because it is non-reentrant, virtual devices that hook interrupts must have some method of synchronizing their calls to the VMM. For this reason, enhanced Windows uses the concept of event processing.

Event procedures are registered asynchronously and, then, called back just before the VMM returns to the application. At this point, the event procedure can use all the VMM services.

VxDs can also use event procedures to perform some action on a VM that is not the current VM. Examples of this include restoring the display to a VM when the display focus changes or simulating an interrupt into a VM the next time the VM is scheduled.

There are two types of events: global and VM specific. Global events are processed before returning to a virtual machine regardless of which VM is about to run. VM specific events are only processed when the specified virtual machine is about to run.

Scheduler

When Windows is running in 386 enhanced mode, each application runs in it own virtual machine (VM). Each VM can be given a share of the CPU time. To assign priority among the VMs, the Virtual Machine Manager (VMM) has a Scheduler.

The Scheduler is the part of the VMM that determines which VM gets CPU time. It is divided into two parts. At the lowest level, the Primary Scheduler maintains execution priorities, and the VM with the highest priority is allowed to run. VxDs will raise and lower the execution priorities to affect task switching among the VMs. The second level of scheduling is handled by the Time Slicer, which boosts a VM's execution priority for a given time slice.

With the Primary Scheduler, there are specific values assigned to execution priorities to accomplish task switching without violating the need for some sections of code to execute exclusively until completion. Additionally, high-priority device events, such as interrupts that must be serviced in a timely manner, will boost execution priorities of VMs that need to be serviced. The VMM provides services and defines execution priorities to handle these cases.

The enhanced Windows Time Slicer is the preemptive multitasking portion of the Scheduler. It relies on time-slice priorities and flags to determine how much CPU time should be allocated to various virtual machines.

Every VM has a foreground and a background time-slice priority. These should be distinguished from a VM's *execution* priority. The VM with the largest execution priority will run, preventing other VMs from executing. The VM with the largest time-slice priority will run more often than other VMs but it will not necessarily prevent other VMs from executing.

Transitions Into and Out of the VMM and VxDs

The enhanced Windows VMM uses the protection mechanism of the 80386 to force privilege ring transitions, as shown in Fibure 16.6 whenever an application program issues a software interrupt or causes a protection fault. One example is when a VM performs I/O to a hooked port. The exact mechanisms used to make the transition into the VMM are not important to a virtual device developer. It is almost never necessary to directly intercept a processor fault or hardware interrupt. The only device that handles hardware interrupts directly is the Virtual PIC (Programmable Interrupt Controller) Device. Callback procedures have been provided to signal a calling routine when a specific event occurs. (See Section 16.5.4, "Callback Procedures," for more information.)

Programmers familiar with the 80386 architecture may assume that, to hook an interrupt, a virtual device will hook the protected-mode Interrupt Descriptor Table (IDT) directly. However, this is not true for Windows in 386 enhanced mode. Services to hook interrupts at this level are provided by the VMM.

WARNING VxDs must never modify the actual IDT. To do so will cause enhanced Windows to crash.

The sequence of events for entering the VMM from a virtual machine because of an interrupt is as follows:

1. The VM performs an operation that generates a fault.
2. A ring transition occurs, and the appropriate IDT interrupt handler is called.
3. The VMM dispatches the interrupt to the appropriate handler by a CALL.

4. The protected-mode handler processes the fault and executes a near RET.
5. The VMM processes any outstanding events.
6. An IRET is executed that causes a ring transition back to the VM.

Notice that the VMM looks at the interrupt before any virtual devices and immediately before returning to the virtual machine.

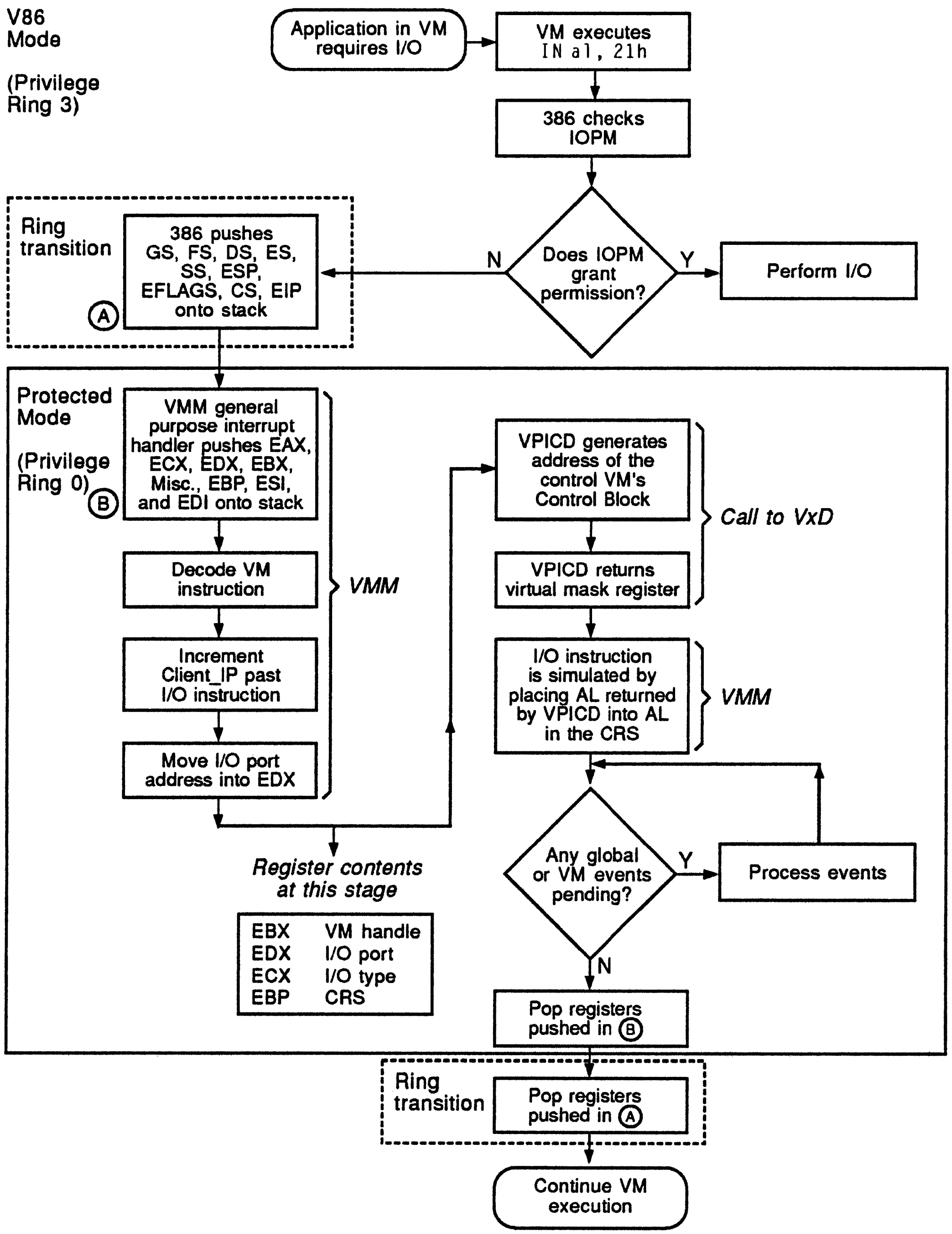

V86 Mode
(Privilege Ring 3)
Application in VM requires I/O
VM executes IN al, 21h
386 checks IOPM
Does IOPM grant permission?
N
Y
Perform I/O
Ring transition
386 pushes GS, FS, DS, ES, SS, ESP, EFLAGS, CS, EIP onto stack
A
Protected Mode
(Privilege Ring 0)
B
VMM general purpose interrupt handler pushes EAX, ECX, EDX, EBX, Misc., EBP, ESI, and EDI onto stack
Decode VM instruction
Increment Client_IP past I/O instruction
Move I/O port address into EDX
VMM
Register contents at this stage
EBX VM handle
EDX I/O port
ECX I/O type
EBP CRS
VPICD generates address of the control VM's Control Block
VPICD returns virtual mask register
Call to VxD
I/O instruction is simulated by placing AL returned by VPICD into AL in the CRS
VMM
Any global or VM events pending?
Y
Process events
N
Pop registers pushed in B
Ring transition
Pop registers pushed in A
Continue VM execution

Figure 16.6

16.5.2 Memory Models

Windows in 386 enhanced mode makes use of the 80386's ability to run different memory models. Some devices may have initialization code that is run in real mode. See Section 16.5.6, "Loading Sequence," for the loading sequence description. After that code is successfully run, a transition is made to protected mode (using selector:offset addressing) in which the VMM is installed and begins executing. The VMM creates a separate VM that consists of a V86-mode portion and an optional, protected-mode (PM) portion for each application.

The VMM and all the enhanced Windows VxDs run in 32-bit, flat-model protected mode. This means that every VxD has complete access to 4 gigabytes of linear address space. A VxD can access any VM's memory at any time.

Because enhanced Windows is flat model, virtual devices cannot change the **CS**, **DS**, **ES**, or **SS** segment registers. These segment registers always contain a selector that has a base of 0 and a limit of 4 gigabytes. Devices can use the **FS** and **GS** segment registers, but there usually is no reason to do so. VMM services will not modify the **FS** or **GS** segment registers. Pointers are always 32-bit linear addresses unless otherwise specified.

NOTE Since the VMM (privilege ring 0 code) resides in a single, flat memory segment, the selector of the selector:offset PM addressing for the VMM and VxDs never changes.

Modes

Application programs typically run in a V86-mode portion of the enhanced Windows operating environment. An example of an exception is the Windows graphic user interface, which also uses a protected-mode portion.

As described in Appendix B, "Understanding Modes," V86 mode is similar to real mode. The crucial difference between the two is that memory protection, virtual memory, and privilege-checking mechanisms are in effect when code runs in V86 mode. Therefore, a program executing in V86 mode cannot interfere with the operating environment or damage other processes. If the program reads or writes memory addresses that have not been mapped into its VM or manipulates I/O ports to which it has not been allowed access, an exception (fault) is generated, and the operating environment regains control.

Privilege Rings

The VMM and the VxDs are at the highest (0) privilege level. Protected-mode applications such as Windows run at privilege level 1, and V86 applications run at privilege level 3, as shown in Figure 16.7.

Since all virtual devices run at protection ring 0, they have the ability to execute any 80386 instruction without producing a protection violation. However, devices should not execute protected instructions as they will usually cause Windows to crash immediately. The only exception to this is the Virtual Math Coprocessor Device, which is allowed to change the 80387 bits in the **CR0** register.

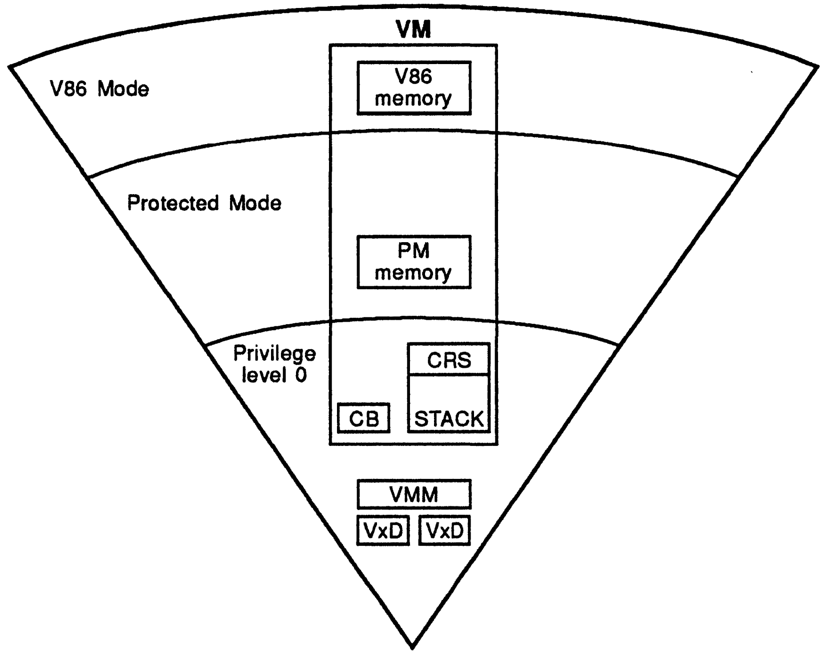


Figure 16.7 Enhanced Windows 3.0 Privilege Rings INTVD_07.EPS

16.5.3 Services

Services are the shared routines of the VMM and VxDs. VxDs use services to handle interrupts, to initiate callback procedures, and to process exceptions/faults.

Notice that there are some VxD services that the VMM requires. Most notable of these are the services provided by the Virtual Programmable Interrupt Controller Device (VPICD), which virtualizes the PIC for the VxDs (for requesting interrupts) and the VMs.

Detailed descriptions of each service are provided in Part 4, "Virtual Device Services." The services are also categorized there as follows:

- Memory Management Services
- I/O Services and Macros
- VM Interrupt and Call Services
- Nested Execution Services
- Break Point and Callback Services
- Primary Scheduler Services

- Time-Slice Scheduler Services
- Event Services
- Timing Services
- Processor Fault and Interrupt Services
- Information Services
- Initialization Information Services
- Linked List Services
- Error Condition Services
- Miscellaneous Services
- Shell Services
- Virtual Display Device (VDD) Services
- Virtaul Keyboard Device (VKD) Services
- Virtual PIC Device (VPICD) Services
- Virtual Sound Device (VSD) Services
- Virtual Timer Device (VTD) Services
- V86 Mode Memory Manager Device Services
- Virtual DMA Device (VDMAD) Services

16.5.4 Callback Procedures

Some services allow a calling routine to register a procedure that will be called back when a particular event occurs. Callback procedures are used for maintaining the VM state via I/O and interrupt trapping and synchronizing with the VMM via the event services.

The VMM includes services that allow virtual devices to install callback procedures to do the following:

- Trap interrupts from virtual machines
- Trap I/O to specific ports
- Trap access to memory
- Schedule per-VM or global time-outs
- Schedule per-VM or global events
- Detect when a VM returns from an interrupt or FAR call

- Detect when a VM executes a particular piece of V86 code
- Detect the release of the critical section
- Detect changes to the VM's interrupt enable flag
- Detect task switches

16.5.5 I/O Port Traps

The VMM provides a service called **Hook_IO_Port**. The service takes two parameters: the port to be hooked, and the address of the procedure to be called whenever the port is accessed.

When a VxD calls **Hook_IO_Port**, the VMM sets the appropriate bit in the I/O permission map (IOPM) and registers the procedure. When a virtual machine executes an instruction that reads or writes data from an I/O port, the 80386 looks up the port number in the I/O permission map. If the corresponding bit in the IOPM is set, then the instruction will cause a protection fault that results in calling the registered procedures.

Hardware Interrupt Hooks

The Virtual Programmable Interrupt Controller Device (VPICD) routes hardware interrupts to other virtual devices, provides services that enable virtual devices to request interrupts, and simulates hardware interrupts into virtual machines.

When a virtual device needs to hook a specific IRQ, it must ask VPICD for permission. If another device has already virtualized the IRQ, then VPICD will refuse.

Software Interrupt Hooks

The software interrupt hooks that are unique to the enhanced Windows environment are described in Chapter 20, "I/O Services and Macros," and Chapter 21, "VM Interrupt and Call Services."

16.5.6 Loading Sequence

The following is a generalized description of the loading sequence. Figures 16.8 and 16.9 are an example of a specific loading sequence.

When Windows in 386 enhanced mode is first started, the following happens:

1. The loader loads the VMM and all the specified virtual devices into extended memory.
2. The loader passes control to the VMM initialization routine.
3. The initialization routine completes the initialization of the VMM and calls all the VxD initialization routines.
4. The System VM is created and initialized.

5. The Shell VxD executes Windows.

Each enhanced Windows device can have different sections of code that are executed during various phases of initialization and normal program execution, as shown in Figure 16.8.

The first phase of initialization is load time. During load time, the virtual device can abort the loading of the device, abort the loading of enhanced Windows, specify instance data, and exclude pages of memory from utilization by enhanced Windows. This load time code is in its own segment and run in real mode and, then, discarded. See Chapter 17, "Virtual Device Programming Topics," for details on real mode initialization.

The rest of the virtual device is run in 32-bit, flat-model protected mode and is divided into four parts:

- Initialization code
- Initialization data
- Code
- Data

The initialization code and data are purged from memory after initialization, as shown in Figure 16.9. These segments contain routines and data that are accessed only during the three phases of enhanced Windows system initialization: **Sys_Critical_Init**, **Device_Init**, and **Init_Complete**. Some of the enhanced Windows VMM services are available only during initialization.

The sections of code and data that are not specifically for initialization perform the device virtualization and can provide services for other devices.

Real Mode

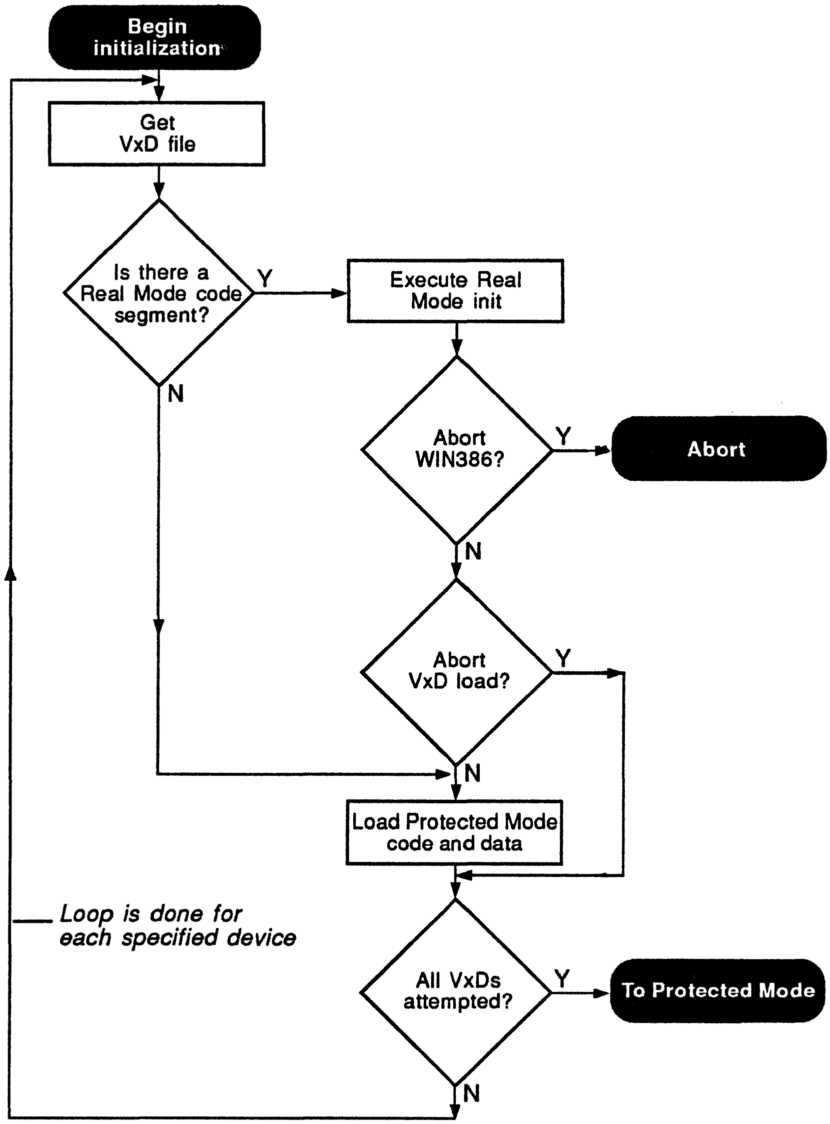


Figure 16.8 The Loading Sequence Flow Chart INTVD_08.EPS

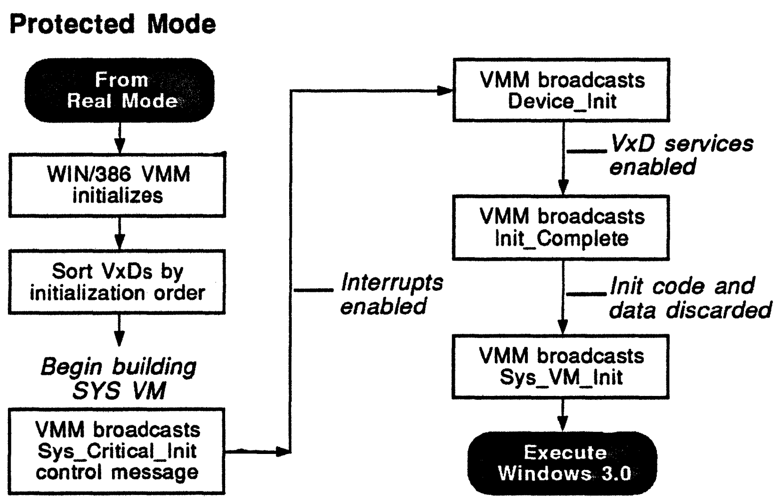


Figure 16.9 The Loading Sequence Flow Chart (cont.) INTVD_08a.EPS

16.5.7 VxD Examples

Often, new VxDs are actually modifications of existing ones. To help with your VxD development, Microsoft includes with the DDK the code for the following fully operational VxDs. We encourage you to use them as examples whenever convenient.

Virtual COM Device (VCD)

The VCD does the following:

- Raises a contention if two VMs access the same port.

Virtual Mouse Device (VMD)

The VMD does the following:

- Reflects mouse interrupts to the VM currently using the mouse.
- Tracks the cursor state at the INT 33H level.

Virtual Printer Device (VPD)

The VPD does the following:

- Raises a contention if two VMs attempt to use the same LPT port.

Virtual Programmable Interrupt Controller Device (VPICD)

The VPICD does the following:

- Virtualizes the PIC I/O for each VM.
- Provides interrupt handling.
- Provides services for other devices to do interrupt handling.

Virtual Sound Device (VSD)

The VSD does the following:

- Tracks the state of the speaker enable bit.
- Times out sound for non-exclusive VMs.

Virtual Timer Device (VTD)

The VTD does the following:

- Queues timer interrupts for each VM.
- Determines which VMs will receive timer interrupts.
- Tracks VMs changing the timer characteristics and may crash them.
- Informs the VMM about elapsed time.

Chapter 17 Virtual Device Programming Topics

This chapter presents details on writing and installing VxDs. You should be familiar with Chapter 16, "Overview of Windows in 386 Enhanced Mode," before proceeding with the material. For explanations on specific types of services provided by the Virtual Machine Manager (VMM), refer to the chapters in Part 4, "Virtual Device Services." This chapter is divided into the following general topics:

- Writing VxDs
- Adding a VxD to Windows
- Initializing a VxD
- Tracking the VM states
- Exiting Windows

We recommend that you scan all the topics before beginning a VxD project. You should also review the sample VxDs supplied on the *Microsoft Windows Device Development Kit* (DDK) disks for examples of how to accomplish specific tasks. The following table suggests some VxDs to study when investigating specific service topics.

Service topic	Sample VxD
Memory management	VDD
Hardware interrupts	VKD
I/O	VPD
Scheduler	VKD
Events	VKD
Timeouts	VKD

17.1 Writing VxDs

Enhanced Windows virtual devices are not "Windows" programs. You do not need to know anything about Windows programming to write a VxD.

Often, new VxDs are simply modifications of existing ones. To help with your VxD development, Microsoft includes the code for many, fully operational VxDs in the *Microsoft Windows Device Development Kit.* We encourage you to use them as examples whenever convenient.

However, some VxDs will require a significant effort to develop. The following can be used as a guideline when writing a complex VxD.

1. Build a skeleton. Using the supplied sources as a guide, build a skeleton of the VxD with the device control procedure, the services, and the API procedures defined but not functional.
2. Add the initialization functionality, including the control block and global memory allocation, physical page hooking, I/O hooking, and interrupt hooking.
3. Fill out the procedures that handle the various hooks.
4. Test the procedures.
5. Implement the APIs and services, if there are any.
6. Test the APIs and services.

17.1.1 Understanding the Ring 0 Memory Model

The part of the enhanced Windows environment containing the VMM and all the VxDs (ring 0), is one, flat-model, 32-bit segment. This means that all the code and data belong to the same group. Two selectors are created: one for code and one for data. Both have a base of zero and a limit of four gigabytes, so all the segment registers point to the same address space (the entire virtual address space provided by the 80386 processor).

When a VxD is loaded, all the offsets are fixed according to the the VxD's actual position. This is different from MS-DOS's loading of .EXE files, in which segments are fixed up and offsets are left untouched.

All procedures are NEAR, and data pointers are 32-bit offsets.

VxDs do not externalize routines or data. To access VMM or VxD services, a dynamic-link mechanism is employed using macros contained in VMM.INC. The VMM services are available with the **VMMcall** macro, and the VxD services with the **VxDcall** macro. Data is shared via declared services only.

You must use the **OFFSET32** macro in your flat model 32 bit segments anywhere you would normally use the **OFFSET** assembler directive. That is, in all segments except for the real-mode initialization segment. This macro correctly defines all the offsets so that LINK386 will do the correct offset fixups. For example:

```
mov   esi, OFFSET32 My_Data
```

17.1.2 VxD Segmentation

As discussed in Chapter 16, "Overview of Windows in 386 Enhanced Mode," VxDs have five functional parts. Each of these parts exists as a separate segment. Macros have been created to define segments for each of the parts.

Each macro name consists of a segment descriptor followed by "_SEG," which means that this macro begins the segment. A segment descriptor terminated by "_ENDS," is used for macros that end the segment. For example, macros used for defining a segment for real-mode load-time initialization would appear as **VxD_REAL_INIT_SEG** and **VxD_REAL_INIT_ENDS.**

In some enhanced Windows installations, it will be possible to demand page portions of VxDs. These installations require a dedicated swap device or a fully virtualized hard disk with a dedicated swap partition. This way, paging can be done without concern for reentering portions of DOS, device drivers, or BIOS. To support paging, a VxD must place the following in locked memory:

- Device Control Procedure (DCP)
- Device Descriptor Block (DDB)
- Hardware interrupt procedures (and the data accessed by them)
- Asynchronous services that can be called from hardware interrup procedures

Some of the macros supplied in VMM.INC (e.g., **Declare_Virtual_Device**) correctly place code and data objects in locked segments. The following are the different segment descriptor types:

```
VxD_REAL_INIT          - Real-mode load-time initialization
VxD_ICODE              - Protected mode initialization code
VxD_IDATA              - Protected mode initialization data
VxD_LOCKED_CODE        - Code that cannot be paged
VxD_LOCKED_DATA        - Data that cannot be paged
VxD_CODE               - Pageable code
VxD_DATA               - Pageable data
```

17.1.3 VxD Declaration

A VxD's first few lines of code must always be the assembler directive, the INCLUDE files, and the declaration parameters.

Assembler Directive

Every VxD must inform the assembler that the code is 80386 protected-mode code. This is done by including the following directive:

```
.386p
```

INCLUDE Files

INCLUDE files enable VxDs to use code located in other parts of enhanced Windows. The following INCLUDE files should always be included:

Filename	Description
VMM.INC	Contains definitions of all the enhanced Window services, as well as required macros and equates.
DEBUG.INC	Contains useful macros for dumping messages to a debugging terminal and performing checks on various data. The macros provided by this file produce code only when the VxD is assembled with the DEBUG switch. See the *Microsoft Windows Software Development Kit* (SDK) for information on the Windows debugging services.
VPICD.INC	Contains equates and service declarations for the Virtual Programmable Interrupt Controller Device (VPICD). All enhanced Windows interrupts are handled by the VPICD. The VPD uses the VPICD services to hook all the printer port's hardware interrupts.
SHELL.INC	Contains definitions of the public services provided by the Shell VxD. The Shell device provides the VxDs with access to the Windows graphics user interface, thus giving the VxDs the ability to display dialog boxes to the user. For example, if two VMs attempted simultaneously to use the same printer, the VxD could call **Resolve_Contention**, which would display a dialog box asking the user to choose between the two VM applications.

Declaration Parameters

The declaration of the VxD is accomplished by its Device Descriptor Block (DDB). The DDB is generated automatically by the **Declare_Virtual_Device** macro. The following example is from the VPD sample provided with the DDK.

```
DECLARE_VIRTUAL_DEVICE VPD,3,0,VPD_Control, VPD_Device_ID, VPD_Init_Order,,,,
```

The table in Figure 17.1 describes each of the parameters:

Parameter		VPD Example	
Name	up to 10 characters	VPD	Required
Major Version	byte number	2	Required
Minor Version	byte number	0	Required
DCP Name		VPD_Control	Required
Device ID	declared in VMM if VxD provides services	VPD_Device_IO	Optional
Initialization Order	determines the order of VxD initialization relative to other VxDs	Since VPD does require initializing before any particular VxD, the number in VMM.INC is large	Optional
Service Table Name		VPD does not provide services	Optional
V86 API Procedure Name		VPD does not provide services	Optional
PM API Procedure Name		VPD does not provide services	Optional

Figure 17.1 The VxD Declaration Parameters PRTGO_01.EPS

17.1.4 VxD Services

The functionality a VxD provides, either to the VMM and other VxDs or through them to applications, is always by means of exported services. After defining the service calling conventions, this section then describes how to declare a service, and verify that a VxD is available to provide a service, and provides a comparison of standard vs. asynchronous services.

Service Calling Conventions

All the enhanced Windows services use either a register-based calling convention or a 32-bit C-type calling convention. In general, all the VMM calls use C calling conventions, and all VxD services are register based.

The C convention services all begin with an underscore (_) in front of the service name. They are similar to the standard C conventions: all parameters are passed on the stack, and results are returned in the **EAX** and **EDX** registers.

Unlike the standard C conventions, the **EBX**, **ES**, **FS**, and **GS** registers are preserved as well as the **ESI** and **EDI** registers. Only the flags and the **EAX**, **ECX**, and **EDX** registers are modified.

The **VMMcall** and **VxDcall** macros support stack parameter passing like the standard C macro package. For example:

```
VMMcall _HeapAllocate, <SIZE Data_Node, 0>
```

will generate the following code:

```
push    0
push    SIZE Data_Node
int     20h
dd      _HeapAllocate
add     esp, 2*4
```

Notice that the parameters are pushed on the stack from right to left as in the standard convention.

All the Windows services for running in 386 enhanced mode that do not begin with an underscore (_) are register-based services. All the parameters to the services are passed in registers and all the results are returned in registers. If a service does not explicitly return a result in a register, than that register will be preserved.

Declaring Services

Virtual devices use two macros, **Begin_Service_Table** and **End_Service_Table**, that are declared in VMM.INC to export services. The service table is normally declared in an INCLUDE file that other VxDs can include to import the services. For example, a typical service table declaration would look something like this for the Virtual "FOO" Device:

```
Begin_Service_Table VFooD

        VFooD_Service vFooD_Get_Version, LOCAL
        VFooD_Service vFooD_Do_Something
        VFooD_Service vFooD_Do_Somthing_Else
        VFooD_Service vFooD_Do_Yet_Another_Thing, VxD_INIT

End_Service_Table VFooD
```

The **Begin_Service_Table** macro uses a single argument to generate the macro used to declare individual services. **Begin_Service_Table** names the macro by taking the name of the device and appending "_Service" to it. In the preceding example, **VFooD_Service** is the name of the macro.

The ***Device*_Service** macro can take one or two parameters. The first parameter is the name of the service (e.g., **Get_Version**). This must match the name of a procedure that was declared with the **BeginProc** macro using the "**Service**" or "**Async_Service**" options. The second parameter is optional. If it is omitted, then the service procedure is declared as an external reference in the VxD_CODE segment.

If the special value "LOCAL" is used as the second parameter (as in the VFooD_Get_Version declaration), then the procedure is not declared as external. This option is used when the service is declared in the same file in which the service table will be created. If, in this case, it were to be declared external, then MASM would generate an error.

If the service procedure is not in the same file as the one used to create the service table, and not in the VxD_CODE segment, then you must supply the name of the segment it resides in so that the proper external declaration can be made. In the above example, the **VFooD_Service VFooD_Do_Yet_Another_Thing** service is declared to be in the VxD_INIT code segment.

The first service for every device *must* be a **Get_Version** service. This service must return with **AX** != 0 and the Carry flag clear. See the following section, "**VxD Presence Verification,**" for more details.

Once the table of services has been created, you must force the table to be generated in one of the VxD source files by defining a special equate (EQU) called "**Create_xxx_Service_Table,**" where xxx is the name of the device before including the service declaration INCLUDE file. For example, the main source file of the VFooD service table would contain the following INCLUDE statements:

```
INCLUDE VMM.INC
INCLUDE Debug.INC
Create_VFooD_Service_Table EQU true
INCLUDE VFooD.INC
```

This must be done in the same source file that contains the device declaration. This table is automatically generated and the pointer to the table is stored in the device's DDB.

Notice that, since the macros generate equates, you will now want to add service declarations to the end of the INCLUDE file. However, never change the order of the declarations. Adding, removing, or changing the order of services changes the service numbering and all the devices that call these services will need to be rebuilt.

VxD Presence Verification

Many devices, such as the EBIOS device, will not load under certain circumstances (for example, when the machine does not have an extended BIOS data area). Before calling device services for devices other than VPICD, Shell, VKD, or other standard devices, you should make sure the device is loaded by calling the device's **Get_Version** service. **Get_Version** for a device will return with **AX** = 0 and the Carry flag set if the device is not installed.

Standard Vs. Asynchronous Services

Most services are not reentrant. This means they cannot be called from hardware interrupt procedures. However, a select group of services is declared as "Async" services and can be called from hardware interrupt procedures. You may declare services that can be called from interrupt handlers by using the "**Async_Service**" option for the **BeginProc** macro.

17.1.5 VxD APIs

While device services are used to communicate with other enhanced Windows virtual devices, APIs are used to communicate with software running in a virtual machine. For example, the Shell device supports an API that is used to communicate with the Windows support program for non-Windows applications that runs in the System VM.

A device can support an API for V86-mode code, protected-mode code, or both. The procedure entry point(s) for the API is specified in the device declaration macro (see Section 17.1.3, "VxD Declaration" for more details on **Declare_Virtual_Device**). The VM software issues an Int 2FH with **AX** = 1684H and **BX** = Device_ID to get the address to call to access the API. See Appendix D, "Windows INT 2FH API," for more information.

When the device API procedure is called with the following parameters:

```
EBX = Current VM handle
EBP = Client register structure
Client_CS:IP = Instruction following API call
```

API procedures must examine the client registers (through the client register structure) to determine which API call was made. The normal calling convention uses **AH** = Major function number and **AL** = Minor function number. Other registers are used for parameters to the functions. However, a device can use any calling convention that is appropriate. If you wish to return a value to the caller, then the API procedure should modify the client registers.

API procedures may modify the **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, and **EDI** registers.

17.2 Adding a VxD to Windows

This section describes in general the steps necessary to install a newly written and debugged VxD into the enhanced Windows environment. These steps are specified and executed from the MAKE file. Detailed instructions are also included in the MAKE file located on the supplied DDK disks.

There are three required steps for installation, with each requiring a specific software tool:

1. Assemble the VxD code with MASM5.10B, which is the special version of the assembler used to handle a new pseudo group, FLAT.
2. Link the .DEF files with LINK386, which is the linker used to create the special 32-bit .EXEs.
3. Declare the code to be a VxD with ADDHDR, which adds special VxD information into the .EXE produced with Link386.

   An optional fourth step, is available for debugging:

4. Generate symbol files with MAPSYM32, which is available to generate 32-bit symbol (.SYM) files for debugging.

These four tools are included in this version of the DDK.

See the following sections for detailed invocation instructions.

The following MAKE file sample is from the Virtual Printer Device (VPD). The complete source for building the VPD is included on the DDK disks.

```
vpd.obj: vpd.asm
         masm5 -p -w2 vpd;

vpd.386: vpd.obj vpd.def
         link386 spd, spd.386/NOI /NOD /NOP,/MAP,,vpd.def
         addhdr vpd.386
         mapsym32 vpd
```

The MAKE file assumes that the four tools are located under the MS-DOS **PATH** command. If they are not, then you must modify the MAKE file to specify their exact locations.

17.2.1 MASM5

This is a special version of MASM that supports 32-bit flat-model code. It has been named MASM5 to differentiate it from other versions of MASM you may already have. It has the same command-line options and format as MASM 5.1, so you can refer to version 5.1 documentation for information on this program.

It is recommended that the **-p** and **-w2** options be used when assembling virtual devices. The **-p** option specifies that impure code segment references should generate warning messages. This is desirable, because it is illegal to write data with a CS override. The **-w2** option sets the warning level to 2, so that warning messages are generated for such things as jumps that are within SHORT range and for data size mismatches.

MASM5 will look for INCLUDE files in the current directory and the INCLUDE path specified by the environment variable INCLUDE. Therefore, the DDK INCLUDE files (e.g., VMM.INC, VPICD.INC, and VDD.INC) should be either in the current directory or located along the INCLUDE path.

17.2.2 LINK386

The LINK386 command line is as follows:

```
link386 <object> {<object>}, <device name>.386 {/<option>}, [<map file
name>][/MAP], ,<device name>.def
```

For example:

```
link386 vpd, vpd.386/NOI /NOD /NOP, /MAP,,vpd.def
```

LINK386 links into one device file the individual object (OBJ) files that make up a virtual device. By convention, Windows devices have the extension .386. The command line specifies the object files(s), the desired output file, option switches, and definition file. The following is a list describing the option switches in the preceding examples.

Option	Full Name	Description
/NOI	NOIGNORECASE	Specifies that case should not be ignored.

Option	Full Name	Description
/NOD	NODEFAULTLIBRARYSEARCH	Specifies that LINK386 should not search for default libraries.
/NOP	NOPACKEDCODE	Specifies that code segments should not be packed into one code segment in the .EXE file.
/MAP		Specifies that all public symbols should be included in the MAP file.

Definition (DEF) files are used with LINK386 to identify the device descriptor block within the device and the types of segments. DEF files for virtual devices all look similar to the following example:

```
LIBRARY VPD

DESCRIPTION 'Win386 VPD Device (Version 2.0)'

EXETYPE DEV386

SEGMENTS
        _LTEXT PRELOAD NONDISCARDABLE
        _LDATA PRELOAD NONDISCARDABLE
        _ITEXT CLASS 'ICODE' DISCARDABLE
        _IDATA CLASS 'ICODE' DISCARDABLE
        _TEXT CLASS 'PCODE' NONDISCARDABLE
        _DATA CLASS 'PCODE'  NONDISCARDABLE

EXPORTS
        VPD_DDB @1
```

The LIBRARY line is required to identify the device as a module that is part of a system rather than an executable application. The DESCRIPTION line is optional and simply records the text string into the .386 file. The EXETYPE line is required to identify the .386 file as an enhanced Windows device file.

The SEGMENTS section is identical for all devices, because it identifies the six possible types of protected-mode segments that can be part of a device. (If a device has a real-mode initialization section, then it can have seven types of segments. However, the real-mode section does not need any special identification in the DEF file.)

The EXPORTS section is also required; it identifies the name and location of the device descriptor block for the virtual device. It must match the name used in the **Declare_Virtual_Device** statement in the device source, with _DDB appended to the end. It must also be identified as ordinal number 1 with the @.

17.2.3 ADDHDR

The ADDHDR command line is as follows:

```
addhdr <device name>.386
```

For example:

```
addhdr vpd.386
```

ADDHDR simply reads the specified 32-bit .EXE file, performs some validation checks, and writes some additional header information needed by the enhanced Windows loader into the file's .EXE header.

17.2.4 MAPSYM32

The MAPSYM32 command line is as follows:

```
mapsym32 <device name>
```

For example:

```
mapsym32 vpd
```

MAPSYM32 reads a MAP file and creates a 32-bit .SYM file for use with the Windows debugger, WDEB386. The **/MAP** option must be specified for LINK386 to generate the necessary MAP file.

17.3 Initializing a VxD

As described in Chapter 16, "Overview of Windows in 386 Enhanced Mode," VxDs are initialized along with the enhanced Windows environment. Both real mode and protected-mode code may be used and are described in the following subsections.

17.3.1 Real-Mode Initialization

Each VxD can have a portion that is run in real mode at load time. This capability is provided to enable a VxD to determine whether or not it can operate in the current environment and to provide information to the loader about how it should vary the environment. This portion is only executed at load time and, then, is discarded.

A real-mode portion is declared as a NEAR procedure in a special 16-bit segment with the rest of the VxD code. At load time, if the loader determines that a real-mode portion is present, it loads it and jumps to its entry point as specified by the END statement at the end of the file (CS:0 if no entry point is found). Upon entry **CS** = **DS** = **ES**, so code and data must be mixed in the same segment. The code can then perform the checks that are necessary and return an exit code back to the loader.

Valid exit codes are as follows:

- 0 - Everything is fine, and the loader should continue loading the protected-mode portion and the rest of the VxDs.
- 1 - This device is not compatible with the current environment and will not be loaded, but the loader can continue to load other VxDs.
- 2 - Something is wrong and the loader should abort the Windows load completely.

If 1 or 2 is returned, then the loader will normally print an appropriate error message naming the VxD that failed. If the real-mode portion has already handled the message reporting or does not want any default error message, then it should set the high bit of the return code in **AX** (i.e., 8001H or 8002H.)

The real-mode portion can also inform the loader to do the following:

- Pass a DWORD of reference data to the protected-mode portion of the device.
- Pass a table of pages in low memory (0-1Mb) that should be excluded from general use by the enhanced Windows memory manager.
- Pass a table of pointers to data that should be instanced for each virtual machine.

It is possible for a VxD to exclude pages and/or declare instance data without actually having a protected-mode portion; it should return 8001H as the return code, so that the loader will attempt no further loading of the device and will not display the default error message.

The real-mode portion can perform most BIOS or DOS interrupts and examine memory to check the environment of the machine. It cannot attempt to perform any type of DOS exit calls because these will halt the loader in an unclean state, and it will be necessary to reboot the machine. Also, any open files should be closed before returning since they will not be closed by the loader.

The following is the actual definition of the real mode initialization interface:

Entry

```
Cs = DS = ES = segment of loaded code and data
IP = specified entry point or 0.
SI = environment segment, passed to the loader from DOS
AX = VMM version number
BX = flags
        bit 0: duplicate device ID already loaded
        bit 1: duplicate ID was from the INT 2F device list
        bit 2: this device is from the INT 2F device list
EDX = reference data from INT 2F response, or 0

SS:SP point to loader's stack.
```

Exit

```
Must return with a NEAR return
AX = return code (see above)
BX = ptr to list of pages to exclude (0, if none), where:
     list
   = one or more words in the range 1 to 9FH (terminated) by a word of zero
SI = ptr to list of Instance data items (0, if none), where:
     list
   = one or more instance data items followed by three words
     of zero (note that 0-3FF, the interrupt vectors are
     always instanced). instance data item = pointer to
     data (word segment, word offset),word length of data

EDX = DWORD of reference data to be passed to the protected-mode portion.
(This can be a linear pointer to ROM data, a constant, etc. that will
affect the way the protected portion might operate. For example, an EBIOS
device can pass the EBIOS page number, so that the protected-mode portion
does not have to look for the page again.)

All the other registers except SS:SP can be modified.
```

The macros **VxD_REAL_INIT_SEG** and **VxD_REAL_INIT_ENDS** are defined in VMM.INC to facilitate creating a real-mode portion of a device driver. The real-mode portion cannot access code or data outside of its segment. If this is attempted, the linker will generate warnings and a corrupt .386 file. Fixed segments such as the BIOS (40H) segments are an exception to this. It is possible to have declared in multiple source files real-mode portions that will all be linked together (e.g., separating message text from the code.)

The following is an example of real-mode initialization code:

```
VxD_REAL_INIT_SEG
BeginProc ebios_init
        mov     ah, 0C0h
        int     15h
        test    es:[bx.SD_feature1], EBIOS_allocated
        jz      short no_ebios_fnd
        mov     ah, 0C1h                        ; get segment adr of EBIOS
        int     15h

        jc      short no_ebios_fnd
        mov     ax, es                          ; get EBIOS segment address
        shr     ax, 8                           ; convert to a page #
        movzx   edx, ax                         ; return EBIOS pg as ref
                                                ; data
        mov     bx, OFFSET exc_ebios_page       ; ptr to exclusion table
        mov     [bx], ax                        ; exclude EBIOS page
                                                ; from memory manager use
        xor     si, si                          ; no instance data to
                                                ; declare
        mov     ax, Device_Load_Ok              ; go ahead and load the
                                                ; device
        jmp     short init_exit                 ; return to loader
```

```
no_ebios_fnd:
        mov     ah, 9
        mov     dx, OFFSET no_ebios_msg         ; print message thru DOS
        int     21h
        xor     bx, bx                          ; no exclusion table
        xor     si, si                          ; no instance data table
        xor     edx, edx                        ; no reference data
        mov     ax, Abort_Device_Load + No_Fail_Message
                                                ; don't load pmode portion
                                                ; and don't display a
                                                ; error msg

init_exit:
ret
exc_ebios_page  dw 0, 0
no_ebios_msg db 'PS/2 type EBIOS not detected', 13, 10, '$'
EndProc ebios_init
VxD_REAL_INIT_ENDS
END ebios_init                          ; specify real mode
                                        ; initialization entry point
```

17.3.2 Protected-Mode Initialization

The enhanced Windows environment has a three-phase, protected-mode initialization. Returning a carry during any of the phases will abort the VxD load.

Phase 1. Sys_Critical_Init

During the first phase of initialization, interrupts are not yet enabled. Therefore, this phase should accomplish the following tasks as quickly as possible.

- Initialization of critical functions necessary when interrupts are enabled.
- Claiming a particular range of V86 pages if necessary (such as the video memory for the VDD).
- Registering device services needed by other devices in later initialization phases.
- Initialization of data needed by the services. During this phase, the System VM **Simulate_Int** and **Exec_Int** commands must not be used.

Phase 2. Device_Init

This is where most devices do the bulk of their initialization. The System VM has been created so interaction with the System VM via such commands as **Simulate_Int** and **Exec_Int** is allowed. Notice that this is the phase where the equivalent functions to **Create_VM** for the System VM. Most VxDs will allocate their control block area or other pieces of memory needed, hook interrupts, hook I/O ports, specify instance data, and initialize themselves and the System VM control block.

Phase 3. Init_Complete

This is the final phase of **Device_Init** that is called just before the WIN386 INIT pages are released and the instance snapshot is taken. VxDs that want to search for a region of V86 pages = A0H to use should do so during this phase. Most devices, though, will not need to do anything here.

17.4 Tracking The VM States

Most likely, the VxD that you are writing needs to keep track of the status of the different VMs that may need your VxD. This includes VM creation, initialization, and termination. The following subsections describe these and other possible VM states.

17.4.1 VM Creation and Initialization

Like the initialization of the enhanced Windows environment, a VM's go through a multi-phase process.

Phase 1. Create_VM

This call creates a new VM. **EBX** = VM handle of the new VM. Returning Carry will fail the **Create_VM**. VxDs should initialize data associated with the VM, especially the control block.

Phase 2. VM_Critical_Init

EBX = VM handle of the new VM. Returning Carry will cause the VM to **VM_Not_Executeable**, then be destroyed. VM **Simulate_Int** or **Exec_Int** activity is allowed. The VxD interacts with the VM to initialize the state of the sofware in the VM (e.g., the VDD does INT 10H to set the initial display mode).

Sys_VM_Init

Same as **VM_Init**, except is initializes the System VM. If Carry is returned, all of enhanced Windows will exit.

17.4.2 VM State Changes

During the normal execution of enhanced Windows, VMs will go through state changes. Most state changes may be ignored by VxDs. However, depending on the purpose of the VxD, some may require VxD response. The following calls describe the possible VM state changes.

VM_Suspend

The VM is not runnable until a resume. **EBX** = VM handle. The call cannot be failed. The VxD should unlock any resources associated with the VM.

VM_Resume

The VM is leaving a suspended state. **EBX** = VM handle. Returning a carry fails and backs out of the resume. Unlock any resources and otherwise prepare internal data structures for the VM to start running again.

Set_Device_Focus

This sets the focus of the specified VxD to the specified VM. **EBX** = VM handle of desired VM. **EDX** = Device ID. If VxD specific set focus, = 0 if device critical set focus (all devices).

This call cannot be failed. Restore the hardware associated with the device to the state of the specified VM. As much as possible, remove VxD interaction with VM (such as disabling I/O trapping) so that VM can run as fast as possible.

Begin_Message_Mode

This call prepares the device for message processing. This is only of interest to the keyboard, mouse, and display. When in message mode, special services provided by the display and keyboard are used to interact with the user. Message mode is used for the Alt+Tab screen and for message boxes when Windows is not available to process a message box. **EBX** = VM handle going into message mode. This call cannot be failed.

End_Message_Mode

EBX = VM handle leaving message mode. This call cannot be failed.

Reboot_Processor

This call requests a machine reboot. The device (usually the keyboard device) that knows how to reboot the machine does the necessary operations.

Query_Destroy

This call asks if it can destroy the running VM. **Query_Destroy** is an information call made by the Shell device before an attempt is made to initiate a destroy VM sequence on a running VM that has not exited normally. **EBX** = VM handle. Returning carry indicates that a device "has a problem" with allowing this. It is recommended that the VxD returning the Carry indicating a problem call **SHELL_Message** to post an informational dialog about the reason for the problem.

Debug_Query

Debug_Query is a special call for device-specific DEBUG information display and activity. This call is made in response to the user typing <VxD name> at the debug prompt, where <device name> is the name specified in the **Declare_Virtual_Device** macro (i.e., in the DDB).

17.4.3 VM Termination

Graceful termination of a VM occurs in the following three steps:

Phase 1. VM_Terminate

During this phase of normal VM termination. **EBX** = VM handle. Call cannot be failed. VM **Simulate_Int** and **Exec_Int** activity is allowed.

Sys_VM_Terminate

Same as **VM_Terminate**, except terminates the System VM (Normal enhanced Windows exit *only*. On a crash exit, this call is not made). System VM **Simulate_Int, Exec_Int** activity is allowed.

Phase 2. VM_Not_Executeable

During the second phase of VM termination. **EBX** = VM handle, **EDX** = Flags (see VMM.INC). Notice that in the case of destroying a running VM, this is the first call made (i.e., the **VM_Terminate** call does not occur). Call cannot be failed. VM **Simulate_Int** and **Exec_Int** activity is *not* allowed. Flags for **VM_Not_Executeable** control call (passed in **EDX**) are as follows:

Flag	Meaning
VNE_Crashed	VM has crashed.
VNE_Nuked	VM was destroyed while active.
VNE_CreateFail	Some device failed **Create_VM.**
VNE_CrInitFail	Some device failed **VM_Critical_Init.**
VNE_InitFail	Some device failed **VM_Init.**

Phase 3. Destroy_VM

During this final phase of normal VM termination. **EBX** = VM handle. Notice that considerable time can elapse between the **VM_Not_Executeable** call and this call. Call cannot be failed. VM **Simulate_Int** and **Exec_Int** activity is not allowed.

17.5 Exiting Windows

There are two calls that can alert a VxD that enhanced Windows is exiting: **System_Exit** and **Sys_Critical_Exit.**

System_Exit

This call is made when Windows is exiting either normally or via a crash. Interrupts are enabled. The instance snapshot has been restored. System VM **Simulate_Int** and **Exec_Int** activity is not allowed. However, the VxD may modify the System VM memory to restore the system state to allow a graceful exiting of Windows.

Syst_Critical_Exit

This call is made when enhanced Windows is exiting either normally or via a crash. Interrupts are disabled. System VM **Simulate_Int** and **Exec_int** activity is not allowed. VxDs should reset their associated hardware to a quiescent state to allow a graceful return to real mode.

Chapter 18 The VDD and Grabber DLL

This chapter describes the Virtual Display Device (VDD) and the Grabber DLL, a Windows dynamic-link library. Software writers should be familiar with the terms and concepts covered in Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," before continuing with this chapter.

The topics in this chapter are presented in the following order:

- Introduction to VDDs
- Converting your 2.x VDD
- VDD device control procedure
- VDD services
- The Grabber DLL and its procedures

18.1 Introduction to VDDs

There are two parts that are necessary to support a video adapter running in 386 enhanced mode.

- The Virtual Display Device (VDD) is the part of Windows in 386 enhanced mode that supports saving, restoring, and emulating the hardware for an application running in a Virtual Machine (VM).
- The Grabber DLL is a Windows dynamic-link library that the WINOLDAP Windows application uses to look at and obtain the state of a VM's video adapter. The Grabber's primary responsibility is rendering the video display into a format that Windows can use.

The general structure of the virtual devices under Windows 3.0 is quite different from the version 2.x structure. While the low-level save, restore, and trapping routines that were written for version 2.x VDDs should work, the interface with the rest of the enhanced Windows environment will be quite different.

The Grabber DLL's interface with WINOLDAP and the VDD is also different. The biggest difference is that in version 2.x the Grabber was WINOLDAP.GRB and not a DLL.

18.1.1 VDD Messages

The sample VDD makes use of Shell event services to keep WINOLDAP informed of changes in a windowed VM's display. When the VDD detects a change in the video state, it sends a message to WINOLDAP, which then queries the state change and modifies the windowed display appropriately.

When the VDD encounters a situation that requires a user's choice or interaction, it uses the Shell message services to print messages and get responses. For example, when there is not enough memory to save and restore a VM's video state, the user is informed of the problem and that a portion of the display may be lost.

18.1.2 VDD I/O Trapping and Hooked Pages

When an application is running in the background, the VDD traps all the video I/O, saving the output port values and emulating the input port values. In some cases, the detection of a mode change can result. In this case, the memory should be disabled and hooked to enable the page fault routine to remap the memory.

A VDD should detect mode changes and illegal memory accesses. This is done by disabling and hooking page faults that occur when the video memory is accessed by the VM. The page fault routine determines how to map the accessed memory by both determining whether the VM has the display focus and by examining the state of the controller. The page fault routine can also be used to demand page the video memory. It will restore and map the video pages needed to create the physical display and to satisfy the application's video memory accesses.

18.1.3 VDD Efficiency

To maximize the efficiency of Windows, a VDD is, in many cases, tightly coupled with the Windows 3.0 display driver. For instance, the EGA display would normally have to be trapped at all times to maintain the controller state properly. Instead, an API has been defined for communications between the Windows display driver and a VDD. Additionally, the EGA Windows display driver uses a special portion of video memory and a special algorithm that allows for a subset of the video controller state to be saved and restored without explicitly saving away the current register values. When adapting a VDD to new displays, it is a good idea to look at alternatives to trapping all the display adapter access to maintain the video state. Notice also that the Grabber is usually tightly coupled to the Windows display driver, specifically to the display-dependent bitmap format.

There are also three PIF bits that the user can specify to disable trapping in VMs where the applications running in the VMs only modify registers that can be read. The VDD designer should use these PIF bits, if possible.

Another good area to consider optimizing API emulation , especially the INT 10H Write TTY function. The user can specify this emulation with a PIF bit.

18.1.4 VDD Development Sequence

Chapter 17, "Virtual Device Programming Topics," discusses the general requirements for writing a VxD. This chapter focuses on the specific example of how to develop a Virtual Display Device. To develop a VDD, follow these steps:

1. Build a skeleton. Using the supplied sources as a guide, build a skeleton of the VDD with all the services and API procedures defined but not functional.
2. Add the initialization functionality, including the control block allocation, global memory needed, physical page hooking, I/O hooking, and interrupt hooking.
3. Fill out the routines that handle the various hooks.
4. Test it while running Windows and other VMs, full screen.
5. Implement the Grabber API, including the procedures that report controller state, return video memory structures, and report video state modifications.
6. Test it while running VMs in a window. Do a thorough test, running many different applications in all the different states (i.e., exclusive, background, and windowed).

18.2 Converting Your 2.x VDD

The core of your Windows 2.x VDD should work with little change. You only need to change some of the way that you access memory. For example, use the **_MapPhysTo-Linear** function rather than adding **PhysToLinr** to physical addresses, and use the control block value **CB_High_Linear** to add to BIOS memory address for accessing those memory locations.

However, you need to do quite a bit of work to change the initialization and system function interface. Additionally, you no longer link VDD with the rest of Windows in 386 enhanced mode, but rather create a separate .386 file that is linked dynamically with the Windows function. It will probably work best if you pull out your Windows 2.x routines and insert them in the Windows 3.0 VDD model sources.

Notice that the definition of exclusive is different for Windows 3.0 and that the **SetFocus** routine takes into account whether or not the VM is running in a window (i.e., VDD will get a **SetFocus** call for the System VM on a VM that is running in a window, instead of a **SetFocus** to the VM itself).

18.2.1 INCLUDE Files

Most of the modules will only need VMM.INC, VDD.INC, DEBUG.INC, and a device specific INCLUDE file (e.g., EGA.INC). Some modules will also require a file describing the interface between them and some external user of their functions (e.g., VMDAEGA.INC for the Grabber). By changing over to the new INCLUDE files, you will generate several undefined references. Modifying the references to use the equivalent Windows 3.0 functionality is a first step in creating your Windows 3.0 VDD.

18.2.2 Changes to the System, Grabber DLL, and Shell Interfaces

Examine the parts of the supplied Windows 3.0 VDD to understand the new system interface. You will need the **VDD_Init**, **VDD_New**, **VDD_Exit**, **VDD_Destroy**, **VDD_SetType**, and **VDD_SetFocus** routines to use the new device control interface. The functionality of **VDD_Install** should be handled by scheduling VM events. The **VDD_Mem_Check** routine is replaced by the VDD specifically calling the Shell to give the user a message. **VDD_CHK_Device** is also replaced by sending WINOLDAP a message when the display needs to be updated and by scheduling time outs to do the detection. The register values and what you can and cannot do in an I/O trap, page fault, and interrupt trap are also changed. Mostly, there is much more flexibility allowed, and there are changes in register save/restore and parameter passing conventions.

Previously, VDD provided a single **VDD_Control** with various subfunctions. Most of the **VDD_Control** calls are replaced by the device API mechanism. Notice that the way that the routines retrieve the Grabber DLL's registers is different (i.e., by using **EBP** and the **Client_Reg** definitions). Also notice the increased number of functions and other changes in the functionality of the Grabber DLL interface.

The Shell device requires a number of new functions that are implemented as device services. Additionally, the old ID call is device service 0. Please see the source examples and other sections of this document for more information.

18.3 The VDD Device Control Procedure

The Device Control Procedure is the dispatch point for most of the Virtual Machine Manager's (VMM) interaction with the VDD. Some modifications in the following areas will be necessary.

18.3.1 Initialization

The following are the areas in the initialization process where some modifications might be needed.

Real Mode Initialization

Most video adapters will not need any real-mode code to be functional. However, a few developers will want to add some real-mode code to query device state or to reserve portions of memory that may not be touched during the initialization of other devices or used for general system purposes. For example, you may have a memory-mapped interface that will be harmed by other code reading and writing at those addresses (Windows in 386 enhanced mode searches the area between C0000H and EFFFFH for the existence of RAM or ROM). Since all real-mode code is executed during system load, it is recommended that you use it only if the same functionality cannot be accomplished during one of the initialization phases of the protected-mode driver.

Sys_Critical_Init

During **Sys_Critical_Init**, a VDD should allocate its control block data area, register services, allocate address space, allocate memory needed globally, and define any pointers or other data that are required for the VDD functionality. Remember that interrupts are disabled during this call, so keep it as short as possible.

Device_Init

During **Device_Init**, a VDD should initialize its global state (such as which VM is currently attached to the physical display), set up the I/O and interrupt trapping needed, specify instance data and, then, initialize the System VM's control block. As noted in Chapter 16, "Overview of Windows in 386 Enhanced Mode," this initialization call is equivalent to **VMCreate** for the System VM, along with the global device initialization.

Init_Complete

During **Init_Complete**, a VDD should do any consistency checks that have to be done after all the other devices have completed their initialization. Normally, a VDD will not need to do anything with this control call.

Sys_VM_Init

During **Sys_VM_Init**, a VDD should set the initial display-mode System VM, initialize the rest of the control block data for the system VM, and set the display focus to the system VM.

18.3.2 VM Creation, Initialization, Destruction, and State Changes

The following areas will also need some modifications:

- During creation, initialize the control block and allocate any VM specific memory. If the allocation fails, return the Carry flag set to abort the VM creation.
- During initialization, set the video state of the VM, typically by making calls to the Video BIOS and trapping the I/O to set up the video state structure.
- During destruction, deallocate any memory allocated for the VM and make sure there are no pointers left that refer to the destroyed VM.
- The **SetFocus** routine is responsible for giving the specified VM the physical display. Notice that there is display **SetFocus** and critical **SetFocus**. Both should give the physical display to the indicated VM. Also notice that the actual restoring of the physical display should occur by executing the **VDD_Restore** routine as an event.

18.4 VDD Services

When a **Begin_Message_Mode** control call is made, the VDD goes into a special mode that allows the Shell device to use the VDD message services to output text to the screen without changing the VM's video state. When the message is complete, an **End_Message_Mode** control call is made that restores the focus VM to the hardware.

As described in Chapter 17, "Virtual Device Programming Topics," a VxD's services are available to the VMM and other VxDs. The following is a list of the general VDD services.

18.4.1 Grabber API

The Grabber uses Get_Version to verify that it is matched with the correct VDD. Whenever the Grabber needs access to the video memory or the video controller state it queries the VDD. The VDD returns a data structure describing the requested memory or controller state.

Get_Mem is used to get current contents of the video memory while updating the windowed display. Get_GrbMem is used to get a snapshot of the entire screen in response to a ALT + PRTSCN from the user in a full screen VM. Free_Mem and Free_Grab are used to tell the VDD that the grabber is no longer using this memory. Get_State and Get_GrbState return current and grabbed controller states respectively. Get_Mod is used to incrementally update the windowed display. Get_Mod returns a data structure which indicates modifications to the current display. The Grabber DLL will modify only those parts of the window that have changed and then issue a Clear_Mod to inform the VDD that the modifications have been carried out.

In order to make sure that the video memory or state will not change when the Grabber is accessing the memory, the VM should not be running after a Get_Mem or Get_Mod call. The VM can continue to run only after a Free_Mem call or an explicit Unlock_App call from the Grabber.

Service	Description
Get_Version	Currently not used, but if implemented, it should return the version.
Get_Grab_Rtn	Called by the shell to determine which routine to call when the user types the garb screen key.
PIF_State	Called by the Shell after VM creation to indicate the VDD PIF bits for the VM. See the definitions for the bits in VDD.INC. The interaction with the VM should be adjusted (i.e., memory allocation and trapping) according to the PIF bits.

Service	Description
Hide_Cursor	Allows the Shell or other device to inhibit the display of the cursor in a window. It has no effect when the VM is full screen and on the VM's video state.
Set_VMType	Called by the Shell each time the VM's execution state or windowed state is changed. VDD should adjust its internal state, such as initializing the structure needed to maintain the display changes when the VM is windowed.
Get_ModTime	Used by the POLL device to determine if the video state is idle.
Set_HCurTrk	Called by the keyboard device when a user is typing keys. It indicates to pass to the Grabber a flag indicating that the cursor position should always remain on the screen for windowed VMs. Otherwise, the cursor will only be tracked as it moves vertically. This prevents excessive horizontal scrolling of the window.
Query_Access	Called by the muse device when it wants to touch the video memory to touch the video memory to update the cursor. If the access canot be handled by the VDD, it will return false.

Each of the following VDD message services is only functional during Message Mode.

Service	Description
Msg_ClrScrn	Clears the screen to an initial background color.
Msg_ForColor	Sets the foreground color.
Msg_BakColor	Sets the background color.
Msg_TextOut	Outputs a string of characters.
Msg_SetCursPos	Sets the hardware cursor position.

The Grabber uses the VDD **Get_Version** service to verify that it is matched with the correct VDD. The API described below enables the Grabber DLL to operate. The Grabber APIs, **Get_Mem** and **Get_GrbMem**, return a data structure that indicates to the Grabber how to look at the VM's video memory. **Get_GrbMem** is caled after the user has typed ALT + PRTSCN when running a full screen application. It returns a shapshot of the video memory at the time the user typed ALT + PRTSCN. **Get_Mem** prevents the VM from running until **Free_Mem** is called so that the video memory and video state will not change while this memory is being accessed.

Free_Mem and **Free_Grab** indicate that the Grabber DLL is done using the memory data structure and, therefore, the data structure can change as necessary. For a screen grab, this indicates to release the memory allocated. For a normal **Get_Mem**, this indicates to allow the VM to run again.

Get_State and **Get_GrbState** return a data structure indicating the controller state of the VM. The controller state coupled with the memory state allow the Grabber to render the VM's video state into a window or into the Clipboard (grab).

Get_Mod and **Clear_Mod** assist the Grabber DLL in rendering a VM's video state into a window. **Get_Mod** returns a data structure that indicates all the changes made to the video state since the last **Get_Mod** call. The Grabber DLL will then modify only those partrs of the window that have changed. **Clear_Mod** indicates to the VDD that the modification state should be initialized to no modifications.

18.5 The Grabber DLL

The Grabber is a dynamic-link library (DLL) primarily responsible for representing the VM's display state to the Windows display driver. It is the library of procedures used by WINOLDAP, the Windows application responsible for creating, destroying, and changing the state of VMs. WINOLDAP makes private calls to the Shell device, which in turn calls the necessary VMM services. Therefore, it is WINOLDAP, using the Grabber (and through it the Windows display driver), that is actually responsible for windowing the display state of a VM.

Each of the Grabber procedures is a **cProc** and has to be exported. The procedure code can be shared by several instances of WINOLDAP, and therefore, the placement of VM-specific data must be deliberate. The Grabber DLL procedures provide support for the following:

- Screen grabbing
- Marking and selecting
- Painting non-Windows applications in a window
- Doing other miscellaneous functions

The Grabber generates data in the following situations:

- When an Extended Paint structure (EXTPAINTSTRUC) is passed from WINOLDAP.
- When a procedure requires local data. (Local data is maintained on the stack.)

18.5.1 On-Screen Selection Interfaces

The user can make on-screen selections with the keyboard, mouse, or through a hot key. The keyboard or mouse are used only while in a window; a hot key (ALT+PRTSCRN) is used while in full-screen or windowed mode.

The procedures that handle on-screen selections are as follows:

- **BeginSelection**

- **EndSelection**
- **KeySelection**
- **AdjustInitEndPt**
- **MakeSelctRect**

To perform a selection by using the keyboard, the user performs the following steps:

1. Choose the Mark command.
2. Move the cursor to the start point of the selection.
3. Sweep through a selection using SHIFT + DIRECTION keys.
4. Press ENTER to end a selection and copy it to the Clipboard (or ESC to end the selection without a copy).

On choosing Mark from the menu, **BeginSelection** gets called with argument <0,0>.

During the first phase, the cursor is moved to the actual start point. **KeySelection** handles the cursor movement. It returns the new start point every time a DIRECTION key is pressed. Notice that the selection could potentially begin at each cursor position. Therefore, every time the start point is changed, **EndSelection** is called to cancel the previous selection, and **BeginSelection** is called with the new start point.

Once the cursor is positioned at the actual start point, the user sweeps through a selection area using SHIFT+DIRECTION keys. **KeySelection** handles the cursor movement. It returns the new end point of the selection. Now each call to **KeySelection** is followed by a call to **MakeSelctRect** to record the current selection rectangle. On pressing ENTER, the actual end point and the final selection rectangle are established.

Therefore, the last call to **BeginSelection** establishes the actual start point, the last call to **KeySelection** returns the actual end point, and the final call to **MakeSelctRect** records the actual selection rectangle. If only DIRECTION keys are pressed, the user is shifting the start point. If SHIFT+DIRECTION keys are pressed, the user is changing the active end point.

NOTE The start point and the end point of a selection have to be aligned on character boundaries in text mode. In graphics mode, the Grabber chooses some granularity for cursor movement (e.g., DWORD of pixels).

The coordinates of the start point and the end point are given in screen coordinates — a window client area position corrected by the scroll bar position. Client area coorindate = <0,0> corresponds to the screen coordinate <ColOrg,RowOrg>. (ColOrg and RowOrg are available in the extended paint structure.)

18.5.2 Selection Interface Procedures

This section presents descriptions of the Selection Interface Procedures in alphabetical order.

AdjustInitEndPt

Description This procedure adjusts the initial selection end point. To start off, the start point and the end point are the same. (This is how **BeginSelection** records them). On the first SHIFT + DIRECTION key call to **KeySelection**, notice that **KeySelection** returns the wrong end point. This routine returns the correct end point. It returns (X+DELTAx, Y+DELTAy) where <X,Y > is the given end point. DELTAx and DELTAy are as defined in **KeySelection.**

Entry lpPntStruct = EXTPAINTSTRUC

YCoOrd,XCoOrd = (Y,X) point to be adjusted

Exit **DX,AX** = (Y,X) end point adjust down and to right for initial selection.

BeginSelection

Description This procedure starts the selection at the indicated point.

Entry lpPntStruc = EXTPAINTSTRUC

YCoOrd,XCoOrd = (Y,X) screen coord of start pt

Exit [lpPntStruc.SelStruc.SelctSRect] Display rectangle in the EXTPAINTSTRUC selection structure set

ConsSelecRec

Description This procedure makes the display rectangle consistent with the selection.

Entry lpPntStruc = EXTPAINTSTRUC

Exit [lpPntStruc.SelStruc.SelctSRect] Display rectangle in the EXTPAINTSTRUC selection structure set.

EndSelection

Description This procedure stops the selection.

Entry lpPntStruc = EXTPAINTSTRUC

Exit None

InvertSelection

Description This procedure inverts the selection.

Entry lpPntStruc = EXTPAINTSTRUC

Exit **DX,AX** = (Y,X) screen CoOrd of "active" selection endpoint

KeySelection

Description This procedure is for keyboard selection.

Entry lpPntStruc = EXTPAINTSTRUC

StartType = 0 if SHIFT key UP
!= 0 if SHIFT key DOWN

MFunc = 0 To Right
= 1 To Left
= 2 Down
= 3 Up

MFunc = 0 To Right
= 1 To Left
= 2 Down
= 3 Up

Exit **DX,AY** = (Y,X) screen CoOrd of new select end pt

KeySelection responds to DIRECTION keys and SHIFT+DIRECTION keys.

DIRECTION key response: (SHIFT key UP)

```
if (LEFT Key)
  return (X-DELTAx, Y)
```

```
;else if (RIGHT Key)
  return (X+DELTAx, Y)
;else if (DOWN Key)
  return (X, Y+DELTAy)
;else if (UP Key)
  return (X, Y-DELTAy);
```

where <X,Y> is current end point.

DELTAx DELTAy are the font width and height in text mode and some appropriate value in graphics mode.

SHIFT+DIRECTION key response:

Similar to above except <X,Y> is current end point.

MakeSelctRect

Description This procedure sets a new selection. It is called after every call to **KeySelection** in response to SHIFT+DIRECTION key. Given a new end point, it adjusts the new end point to be character-aligned in text mode (and on a convenient boundary in the video memory in graphics mode). It also adjusts for screen maxima. It sets the Selection rectangle based on the current start point and end point.

Entry lpPntStruc = EXTPAINTSTRUC

YCoOrd,XCoOrd = (Y,X) screen CoOrd of new end point

Exit [lpPntStruc.SelStruc.SelctSRect], Display rect in extended paint selection structure set

AX == 0, if no change was made to selection parameters

NOTE [lpPntStruc.SelStruc.SelctSRect] *must still be set* in this case

RenderSelection

Description This procedure renders the selection into the Clipboard format.

Entry lpPntStruc = EXTPAINTSTRUC

wParam	Parameter from VDD message (= -1 if VMDOSAPP origin
lParam	Parameter from VDD message (=0 if VMDOSAPP origin) Event ID

Exit

```
if (DX < 0)
    Error
else if (DX = 0)
   No Selection
else if (DX > 0)
 DX = format, (CF_OEMTEXT or CF_BITMAP)
 AX = Handle, (Memory Handle or Bitmap Handle)
```

18.5.3 Non-Windows Application Painting Interfaces

This section presents descriptions of Non-Windows Application Painting Interfaces in alphabetical order.

GetDisplayUpd

Description

This procedure calls the VDD to get a display update (if any) and stores it in the Paint structure.

It prevents any further changes from occurring in the application. The application restarts after a call to one of the following; UpdateScreen, PaintScreen, or GrbUnLockApp.

Entry

lpPntStruc = EXTPAINTSTRUC

wParam	Parameter from VDD message (= -1 if VMDOSAPP origin)
lParam	Parameter from VDD message (=0 if VMDOSAPP origin) Event ID

Exit

AX = Display update flags (see grabpnt.inc for fDisp_ flags)

PaintScreen

Description

This procedure paints the indicated region of the screen.

This procedure paints the non-Windows application screen into a window. The origin of this is a Windows paint as opposed to a display update (that is handled at **UpdateScreen**). When a non_Windows application receives a Windows Paint message, **Paint Screen** gets called,

Entry

lpPntStruc = EXTPAINTSTRUC

Exit

AX != 0 Screen Painted

AX == 0 Screen not painted, probably low Windows memory problem

SetPaintFnt

Description This procedure sets the font for painting in the extended paint structure so that WINOLDAP can compute the paint rectangle for use on **PaintScreen** calls. This is called right before a call to **PaintScreen**. It is also called right before a call to **UpdateScreen**.

Entry lpPntStruc = EXTPAINTSTRUC

lpWidFullScr	Word pointer for width return
lpHeightFullScr	Word pointer for height return

Exit FntHgt and FntWid values in EXTPAINTSTRUC set

NOTE Values are set to 0 if it is a graphics screen.

[lpWidFullScr] = Width of full screen in pix (Text or Graphics)

[lpHeightFullScr] = Height of full screen in pix (Text or Graphics)

DX is height of full screen in scan lines if Graphics, in text lines if Text.

AX is width of full screen in pix if Graphics, in chars if Text.

UpdateScreen

Description This procedure updates changed portions of the screen. When a non-Windows application modifies the display on its own, **UpdateScreen** is called.

Entry lpPntStruc = EXTPAINTSTRUC

Exit **AX** == 1 if Screen Paint, unless fGrbProb bit set in EPStatusFlags

AX == 0 if Screen not painted, probably low Windows memory problem

GrbUnLockApp

Descripton This procedure undoes the implied application lock of GetDisplayUpd

Entry lpPntStruc = EXTPAINTSTRUC

Exit None

18.5.4 Miscellaneous Interfaces

This section presents descriptions of Miscellaneous interfaces in alphabetical order.

CheckGRBVersion

Description This procedure checks out the VDD version.

Entry lpPntStruc = EXTPAINTSTRUC

Exit

```
If (AX == 0)
        OK
AX != 0
        Bad version
AX == 1
        Version # error
AX == 2
        Display type mismatch (VDD and Grabber are not compatible)
        DX = Grabber Version number
```

CursorOff

Description This procedure destroys the cursor for an application.

Entry lpPntStruc = EXTPAINTSTRUC

Exit Caret destroyed

CursorOn

Description This procedure creates the cursor for an application if it has one.

Entry lpPntStruc = EXTPAINTSTRUC

Exit Caret created

CursorPosit

Description	This procedure returns the position of the cursor on the display.
Entry	lpPntStruc = EXTPAINTSTRUC
Exit	**DX,AX** = (Y,X) screen CoOrd of upper left of cursor = (-1,-1) if no cursor

GetFontList

Description	This procedure returns a pointer to the list of extra fonts you want loaded.
Entry	lpFontBuf -> Buffer for font info
Exit	Font Buffer filled in

GrabComplete

Description	Signals that we are finished with the grab. This is called after the grab is complete. It is time to call the VDD and have it free the grab memory.
Entry	lpPntStruc = EXTPAINTSTRUC wParam — Parameter from VDD message (= -1 if VMDOSAPP origin) lParam — Parameter from VDD message EVENT ID
Exit	None

GrabEvent

Description	Private Grabber messages. This procedure provides a private channel of event communication between the VDD and the Grabber to perform a hot key screen grab.
Entry	lpPntStruc Extended paint structure wParam — Parameter from VDD message

	lParam	Parameter from VDD message EVENT ID

Exit None

InitGrabber

Description This is the library initialization procedure.

Entry **DI** = Module handle of the library

CX = Size of local heap (should be 0)

DS = Seg addr of library data segment (isn't one)

Exit

```
AX == 0
  Init Error
AX != 0
  OK
```

ScreenFree

Description This procedure frees anything associated with this application.

Entry lpPntStruc = EXTPAINTSTRUC

Exit Any allocated stuff associated with the application is freed.

Part 4 Virtual Device Services

This part documents all the enhanced Windows virtual machine environment services. They are grouped by service type and presented in the order shown on the following page.

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

CHAPTERS

Chapter 19 Memory Management Services

Note to Readers: *The introduction for this chapter should be considered potentially inaccurate as it has not been proofed for technical accuracy. However, the individual services documentation may be considered authoritative, though it has not been edited for grammer.*

Enhanced Windows supplies a rich set of memory management services. Since many of the services are unnecessary for most VxD development, only a commonly used subset is listed in this introduction. However, all the memory management services are documented in either this chapter or in Chapter 40, "V86 Mode Memory Manager Device Services."

See also Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions. Memory management is also discussed in the *Microsoft Windows Software Development Kit, Programming Tools* and in Chapter 6, "Network Support," in the *Microsoft Windows Device Driver Adaptation Guide*".

The Enhanced Windows environment uses a virtual memory scheme capable of overcoming the limits of actual physical memory. Though it may not be physically present, a virtual memory of 4 gigabytes is theoretically addressable. This is done by swapping (paging) code and data to and from RAM and a secondary storage device. Since VxDs reside within the 32-bit protected-mode portion of the environment, they may make use of the scheme's advantages by using the memory management services.

Windows determines the amount of virtual memory actually available based on the total amount of physical memory on the system and the amount of disk space available. This can be changed (downward) by modifying the swap file size specified in the SYSTEM.INI file.

Windows will continue to allocate physical memory until it has been used up. Then, it will begin moving 4-kilobyte pages of code and data from physical memory to disk to make additional physical memory available. Windows pages in 4-kilobyte blocks, rather than unequal-sized code and data segments. The swapped 4-kilobyte block may be only part of a given code or data segment, or it may cross over two or more code or data segments.

This memory paging is transparent to a program. If an attempt is made to access a code or data segment of which some part has been paged out to disk, the 80386 issues a page fault interrupt to Windows. Windows then swaps other pages out of memory and restores the pages that the program needs.

The Windows memory management services are presented in the following categories. The services specified under some of the categories comprise the commonly used subset.

- System Data Object Management

  Allocate_Device_CB_Area

- Device V86 Page Management

  Assign_Device_V86_Pages

- GDT/LDT Management
- System Heap Allocator

  HeapAllocate

  HeapFree

- System Page Allocator

  CopyPageTable

  MapIntoV86

  ModifyPageBits

  PageAllocate

  PageFree

  PageLock

  PageUnlock

  PageGetAllocInfo

  PhysIntoV86

- Looking at Physical Device Memory in Protected Mode

  MapPhysToLinear

- Data Access Services

  GetFirstV86Page

- Special Services for Protected Mode APIs
- Instance Data Management
- Looking at V86 Address Space

(Are we missing GetNullPageHandle, AddInstanceItem & LookingatV86Address-Space?)

19.1 System Data Object Management

These services provide support for allocating special system areas. The three areas managed are the Control Block (i.e., the data structure passed to VxDs indicating which VM is involved), the Global V86 Addressable Area, and the GDT and LDT.

NOTE All of these calls use the USE32 C calling convention. The true name of the procedure has an underscore in front (i.e., **Allocate_Device_CB_Area** is actually **_Allocate_Device_CB_Area**), and the arguments are pushed right to left (unlike the PL/M calling convention used by Windows, which is left to right). The return value(s) is returned in C standard **EDX:EAX**. It is the responsibility of the *caller* to clear the arguments off the stack. Registers **EAX**, **ECX**, and **EDX** are changed by calls. Registers **DS**, **ES**, **FS**, **GS**, **EBP**, **EDI**, **ESI**, and **EBX** are preserved.

Allocate_Device_CB_Area

```
unsigned Allocate_Device_CB_Area(nBytes,flags)
unsigned nBytes;
unsigned flags;
```

This call is used to allocate a region of the Per VM Control Block data structure to a particular device. Devices typically want some data that is "per VM". For example, a device which is virtualizing a particular set of I/O ports for the VM needs a place to store each VMs "instance" of the I/O port state. This is done by allocating a region of the VM Control Block large enough to hold a device specific data structure which contains the state. For example, if the device specific data structure looks like this:

```
FooDeviceCB     Struc
        FooDevReg1          db      ?       ; Dev I/O register 1
        FooDevReg2          db      ?       ; Dev I/O register 2
        FooDevReg3          db      ?       ; Dev I/O register 3
        FooDevReg4          db      ?       ; Dev I/O register 4
        FooDevState         dd      ?       ; State flags for device
FooDeviceCB     Ends
```

Space in the VM Control Block would be allocated like this:

```
VxD_DATA_SEG
FooDevCBOffset  dd      ?
VxD_DATA_ENDS
VxD_ICODE_SEG
;
; Allocate the Control Block space. This is in Foo's INIT routine
;
        VMMCall _Allocate_Device_CB_Area,<<SIZE FooDeviceCB>,0>
        or      eax,eax
        jz      short No_CB_Space_Error ; Probably FATAL error
        mov     [FooDevCBOffset],eax

VxD_ICODE_ENDS

VxD_CODE_SEG
;
; In VxD procedures the Control Block pointer is passed
;   in EBX the control block may be pointed to like this.
;
        mov     edx,ebx
```

```
            add     edx,[FooDevCBOffset]
            mov     al,[edx.FooDevReg1]
            ...

VxD_CODE_ENDS
```

The *nBytes* parameter specifies the number of bytes of space to be allocated. There are currently no bits defined in the flags, this parameter must be set to 0.

Return Value

Returns nonzero Control Block Offset of the block allocated if successful, returns zero if the space could not be allocated (This is probably a fatal error, it is up to the caller to decide what is to be done in this case).

Comments

Control block Offsets returned from this call will be DWORD aligned. The *nBytes* parameter does not have to be a multiple of 4, but if it isn't, it will currently be rounded up to a multiple of 4. This may change in a later releases, so do no depending one rounding.

The above code sample is not the only way to do things. There are many other ways the Control Block Offset value can be used to access your devices specific region of the control block.

NOTE This routine itself is in the init segment of WIN386. It can therefore only be called during system initialization. Trying to call it after system initialization and the system INIT segment space has been reclaimed will result in a fatal page fault.

When Control Block regions are allocated they are initialized with value 0 in all bytes. When new VMs are created, all bytes of the Control Block are set to 0.

Allocate_Global_V86_Data_Area

```
unsigned Allocate_Global_V86_Data_Area(nBytes,flags)
unsigned nBytes;
unsigned flags;
```

This call is used to allocate a region of the Global V86 Addressable Area to a particular device. This area is used for device specific objects which must also be addressable by the Virtual mode code running in the Virtual Machine.

An example is a Virtual mode software interrupt which is trapped by the device and causes the return of a Virtual mode pointer to some data associated with the device. The data must be in the VM's V86 address space since a Virtual mode pointer to it is returned. In this case there is no reason for the interrupt hook code to also be in the Global V86 Addressable Area, that can all be in the protected mode device.

The *nBytes* parameter specifies the number of bytes of space to be allocated. Current flags bits:

```
GVDAWordAlign    EQU    00000000000000000000000000000001B
GVDADWordAlign   EQU    00000000000000000000000000000010B
GVDAParaAlign    EQU    00000000000000000000000000000100B
GVDAPageAlign    EQU    00000000000000000000000000001000B
GVDAInstance     EQU    00000000000000000000000100000000B
GVDAZeroInit     EQU    00000000000000000000001000000000B
GVDAReclaim      EQU    00000000000000000000010000000000B
```

All unused bits must be zero. **GVDAxxxxAlign** bits specify the indicated alignment (WORD, DWORD, PARAGRAPH, PAGE) for the start of the block. If none are set, BYTE alignment is assumed. **GVDAInstance**, if set, indicates that the block is an item of VM instance data for which each different VM has its own private values. If **GVDAInstance** is clear, the block is global data and all VMs share the same value setting. **GVDAZeroInit**, if set, indicates that the block is to initialized with value 0 in all bytes of the block. If **GVDAZeroInit** is clear, the block will have random values in it.

GVDAReclaim is only valid if **GVDAPageAlign** is set. IF **GVDAReclaim** is set, then the physical pages of the region should be "reclaimed" by the MMGR and placed on the free list, and the NUL page should be mapped in the region.

Return Value

Returns nonzero linear address of the block allocated if successful, returns zero if the space could not be allocated. This is probably a fatal error, it is up to the caller to decide what is to be done in this case.

Comments

The Flag bit equates are defined by including VMM.INC. The equates should be used.

For blocks allocated with **GVDAInstance** set, the **AddInstanceItem** call is made by this routine for you.

Note the interaction with **Allocate_Temp_V86_Data_Area.**

Specifying multiple **GVDAxxxxAlign** bits will result in random behavior. At most ONE of these bits must be set.

The returned linear address is a ring 0 linear address. It is up to the caller to convert this into a Virtual mode SEG:OFFSET form if that is needed.

The linear addresses returned by this call will be <100000h the limit of virtual mode addressability.

Generally only data needs to be placed in these blocks, but code can be placed if desired.

WARNING You must be *very careful* if allocating two blocks, one for code which is *not* instanced, and one for data which *is* Instanced because you *cannot* assume that the two blocks will be within 64K of each other and thus addressable with the same segment register in virtual mode.

If the VxD desires the values of Instance fields allocated with this call to have a set initial value whenever a new VM is created, the field must be initialized with the desired values immediately after making this call. The contents of the instance blocks at the time VxD initialization is completed is what each new VM is created with.

NOTE This routine itself is in the init segment of WIN386. It can therefore only be called during system initialization. Trying to call it after system initialization and the system INIT segment space has been reclaimed will result in a fatal page fault.

Special notes for GVDAPageAlign

This type of allocation is intended to support Vxds which need a global page aligned piece of V86 address space where they can **MapIntoV86** data. The best example of such a VxD is the **PageSwap** device.

The *nBytes* parameter should be a multiple of 4096 (page size).

Note that this page is global but that **MapIntoV86**, **PhysIntoV86**, and **LinMapIntoV86** are calls which are local to a specific VM. This means that a VxD which wishes to globally change the mapping of this region must traverse the VM list with **Get_Next_VM_Handle** and perform the map in each VM individually.

WARNING Do not issue any of the map calls on this region before SYS_VM_Init device call time. Failure to follow this rule can cause the page type bits in the page table to get set improperly.

VxDs using this should set the correct initial VM state in their **Create_VM** device call code. The initial state of the region is actually a copy of the current state of **SYS_VM_Handle**, but you should not rely on this. Set the initial state you want explicitly by making a **MapIntoV86,** or **PhysIntoV86** call.

The physical page(s) which are mapped into this region at the time you allocate it are not pages that the MMGR worries about. It is up to the VxD to put the physical pages to good use. The addresses of these physical pages(s) is found by doing a **CopyPageTable** call on the **SYS_VM_Handle** and looking at the physical address in the page table entries.

Do not assume that the physical addresses of these pages equals the linear address returned. This will be true on most machines, but not on some. These pages by using are mapped with **PhysIntoV86**.

If **GVDAReclaim** is set, then the physical pages that currently are mapped in the region will be reclaimed by the MMGR and placed on the free list. The NUL page will then be mapped in the region.

If **GVDAReclaim** is clear, the *physical* page(s) which are mapped into this region at the time you allocate it are not pages that the MMGR worries about. It is up to the VxD to use these physical pages for something useful. Try to avoid just wasting them. The addresses of these *physical* pages(s) is found by doing a **CopyPageTable** call on the **SYS_VM_Handle** and looking at the physical address in the page table entries.

It is invalid to assume that the physical addresses of these pages = the linear address returned. This will be true on most machines, but on some it will not. These pages are mapped using **PhysIntoV86**.

You will not be able to **Assign_Device_V86_Pages** the pages of this region. They are already marked as globally owned because they are below **FirstV86Page**.

You cannot set both **GVDAReclaim** and **GVDAInstance**. Attempting to do so will result in an error.

Allocate_Temp_V86_Data_Area

```
unsigned Allocate_Temp_V86_Data_Area(nBytes,flags)
unsigned nBytes;
unsigned flags;
```

This call is used to allocate a region of the Global V86 Addressable Area to a particular device during system initialization.

The primary reason for allocating this area is to create a buffer into which data associated with some **Simulate_Int** activity (like an INT 21H DOS system call) can be placed. The area allocated with this call only exists for a short period of time during initialization. The *nBytes* parameter specifies the number of bytes of space to be allocated. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value

Returns nonzero linear address of the block allocated if successful, returns zero if the space could not be allocated (insufficient memory, or temp area already allocated).

Comments

There is only one Temp area, therefore only one allocation will be allowed to be outstanding at a time. Attempts to allocate the Temp area when it is already allocated will result in an error.

The **Allocate_Global_V86_Data_Area** call does not function while the Temp Area is allocated. The Temp Area must be released with **Free_Temp_V86_Data_Area** before the **Allocate_Global_V86_Data_Area** call can be made again.

Make sure you **Free_Temp_V86_Data_Area** the temp area as soon as possible.

The returned linear address is a ring 0 linear address. It is up to the caller to convert this into a Virtual mode SEG:OFFSET form if that is needed.

The linear address returned by this call will be <100000h the limit of virtual mode addressability.

Since this area exists only temporarily, it doesn't make sense to Instance any of it.

The linear address returned from this call is paragraph aligned.

The contents of the block will always be Zero Initialized by this call.

NOTE This routine itself is in the init segment of WIN386. It can therefore only be called during system initialization. Trying to call it after system initialization and the system INIT segment space has been reclaimed will result in a fatal page fault.

Free_Temp_V86_Data_Area

```
unsigned Free_Temp_V86_Data_Area()
```

This call is used to free the **Temp_V86_Data_Area** allocated with **Allocate_Temp_V86_Data_Area.**

Return Value Returns nonzero if successful, returns zero if unsuccessful (Temp Area not allocated).

Comments The **Allocate_Global_V86_Data_Area** call does not function while the Temp Area is allocated. The Temp Area must be released with **Free_Temp_V86_Data_Area** before the **Allocate_Global_V86_Data_Area** call can be made again.

Once this call is issued, the Linear Address that was returned from **Allocate_Temp_V86_Data_Area** can no longer be used for anything. The system will probably crash if this is attempted.

NOTE This routine itself is in the init segment of WIN386. It can therefore only be called during system initialization. Trying to call it after system initialization and the system INIT segment space has been reclaimed will result in a fatal page fault.

19.2 Device V86 Page Management

Certain types of VxDs may want to “take over control” of certain regions of VM V86 address space for use by the VxD. The best examples of this are as follows:

- The display device (VDD), which wants to reserve those areas of the A0H to BFH page address range that are used by the display device.
- The EMM device (part of V86MMGR), which wants to use a region of VM V86 address space between pages A0H and 100H for the high memory EMM 3.20 Mapping Window.
- The device responsible for management of the EBIOS page, page 9FH, on machines like the IBM PS/2 Model 80.

The following calls enable VxDs to allocate VM V86 address ranges for such purposes and cooperate with other VxDs that also might want to use them. There are two types of assignment that can be used: global, which applies to all VMs in the system, and local, which applies to only one VM. The VDD video and EBIOS page assignments are examples of global assignment (although these could be local depending on the specifics of the implementation). The EMM assignments are an example of local assignments. The EMM driver does not want to take over VM V86 page assignment in VMs that are not using EMM because then all those pages cannot be used by any other device. Thus, it waits until a specific VM makes an EMM call of a certain type at which point the EMM driver *may* do a local page assignment in that particular VM to assign the EMM pages of the V86 address space to the EMM device. The global versus local assignment is specified via the *VMHandle* parameter on the calls. If the handle is nonzero, it is local; if the handle is zero, it is global.

No protection is provided with this mechanism; all that is provided is information so that devices can cooperate. There is nothing to prevent a VxD from mapping pages that it does not own or a page owned by some other VxD. A device that does these things is simply un-cooperative and not correctly implemented.

NOTE All of these calls use the USE32 C calling convention. The true name of the procedure has an underscore in front (i.e., **Assign_Device _V86_Pages** is actually **_Assign_Device_V86_Pages**), and the arguments are pushed right to left (unlike the PL/M calling convention used by Windows, which is left to right). The return value(s) is returned in C standard **EDX:EAX**. It is the responsibility of the *caller* to clear the arguments off the stack. Registers **EAX**, **ECX**, and **EDX** are changed by calls. Registers **DS**, **ES**, **FS**, **GS**, **EBP**, **EDI**, **ESI**, and **EBX** are preserved.

Assign_Device_V86_Pages Assign_Device_V86_Pages service

```
unsigned Assign_Device_V86_Pages(VMLinrPage,nPages,VMHandle,flags)
unsigned VMLinrPage;
unsigned nPages;
unsigned VMHandle;
unsigned flags;
```

This call is used to assign a region of VM V86 address space to a device. *VMLinrPage* specifies the linear page number (>=0, <=10Fh) of the first page of V86 address space to be assigned. *nPages* specifies the number of pages to be assigned starting at *VMLinrPage*. The entire specified range must be >=0, <=10Fh, an error will occur if it is not. All of the specified pages must be un-assigned, or an error will occur. *VMHandle* specifies the VM to Local assign the pages in, if this parameter is 0, it means the pages are to be Global as-signed. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value Returns nonzero if the assignment was successful, returns zero if the assignment failed (at least one page in the specified range is already assigned, or invalid page range).

Comments During device initialization only Global Assignments are allowed, and there are restric-tions on the pages which can be assigned. Pages between **FirstV86Page** and page 0A0h can only be top down, in order assigned during device initialization. Local Assignments, and General assignment between **FirstV86Page** and page 0A0h must wait until device in-itialization is complete.

Note that Global Assignment of a page that is already assigned, either Local to any VM, or Global assigned will fail. Global assignment can only work on pages which are not currently assigned in any VM.

DeAssign_Device_V86_Pages

```
unsigned DeAssign_Device_V86_Pages(VMLinrPage,nPages,VMHandle,flags)
unsigned VMLinrPage;
unsigned nPages;
unsigned VMHandle;
unsigned flags;
```

This call is used to deassign a region of VM V86 address which was previously assigned with **Assign_Device_V86_Pages.** *VMLinrPage* specifies the linear page number (>=0, <=10Fh) of the first page to be deassigned. *nPages* specifies the number of pages to be deassigned starting at *VMLinrPage*. The entire specified range must be >=0, <=10Fh, an error will occur if it is not. All of the specified pages must be assigned, or an error will occur. *VMHandle* specifies the VM to Local deassign the pages in, if this parameter is 0, it means the pages are to be Global deassigned. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value Returns nonzero if the deassignment was successful, returns zero if the deassignment failed (at least one page in the specified range is already deassigned, or invalid page range).

Comments During device initialization this call will always fail. This call only works after device initialization is complete.

An extreme amount of chaos will occur if someone Global DeAssigns a range which is actually Local Assigned, or DeAssigns a region which was not obtained via a successful **Assign_Device_V86_Pages.**

Get_Device_V86_Pages_Array

```
unsigned Get_Device_V86_Pages_Array(VMHandle,ArrayBufPTR,flags)
unsigned VMHandle;
unsigned ArrayBufPTR;
unsigned flags;
```

This call is used to obtain a copy of the assignment bit map array for **Device_V86_Pages.** This allows the caller to determine which regions of the VM V86 address space are currently assigned, and which are available. *VMHandle* specifies the VM to get the assignment bit map of, if this parameter is 0, it means to get the Global assignment array. *ArrayBufPTR* points to a buffer large enough to contain the array. The assignment array is an array of 110h bits, one bit for each page in the range 0-10Fh. Thus the size of the array is ((110h/8)+3)/4 = 9 DWORDS.

Bits in the array which are set (=1) indicate pages which are assigned, bits which are clear (=0) indicate pages which are not assigned. Thus to test the bit for page number N (0 N 10Fh) you could use code like this:

```
mov     ebx, N MOD 32                              ; Bit number in DWORD
        mov     eax, N / 32                        ; DWORD index into array
        bt      dword ptr ArrayBufPTR[eax*4],ebx; Test bit for page N
        jnc     short PageUnAssigned     PageAssigned:
```

Note that this code is mearly intended to illustrate how the bit array works. This code is not the most efficient, or the only way to implement this test. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value Returns nonzero if successful, returns zero if the bit array could not be returned (Invalid *VMHandle*).

Comments The Global Bit Array only indicates those pages which are currently Globally owned. Bits with 0 in them do not necessarily indicate pages which can be **Global Assign_Device_V86_Paged**. The reason is that one of the VMs in the system may have that page Local **Assign_Device_V86_Paged**. In order to determine if a page can be globally assigned, the Global array must be examined, AND all of the VM Local arrays must be examined.

19.3 GDT/LDT Management

These services provide a way for VxDs to allocate Global Descriptor Table (GDT) selectors and set up a Local Descriptor Table (LDT) for protected-mode execution. Notice that the intent of these services is to support segmented environments in protected mode. In general, VxDs should never need to allocate GDT selectors or set up an LDT. The only reason these services are needed is to support protected-mode applications. Notice that the LDT is a per-VM object; each VM can (may) have its own LDT. Since enhanced Windows is a flat model system, do not create multiple segments.

Allocate_GDT_Selector

```
unsigned Allocate_GDT_Selector(DescDWORD1,DescDWORD2,flags)
unsigned DescDWORD1;
unsigned DescDWORD2;
unsigned flags;
```

This call is used to create a new GDT selector. *DescDWORD1* and *DescDWORD2* form the 8 bytes of information to be placed in the new descriptor. *DescDWORD1* is the high order 4 bytes of the descriptor containing the high 16 bits of the base, the high 4 bits of the limit and the status and type bits. *DescDWORD2* is the low order 4 bytes for the descriptor containing the low 16 bits of the base and limit. Use **BuildDescDWORDs** to help you set up these arguments. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value

Returns a 64 bit long which is actually two 32 bit DWORDs. The low DWORD (**EAX**) is the non-zero selector if succesfull. The high DWORD (**EDX**) is split into two 16 bit word returns. The low 16 bits of **EDX** is the GDT descriptor which describes the GDT itself. Unlike the LDT, it is strongly recommended that this selector *not* be used to edit the GDT. If you mess up editing the LDT, you will probably just crash one app, but if you mess up editing the GDT, you will crash the whole system. The high 16 bits of **EDX** is the number of selectors currently in the GDT (the "limit" of the GDT expressed as a number of selectors, (LIMIT+1)/8). Both DWORDS have value 0 if the allocation failed (Bad DescDWORD arguments, GDT is full, insufficient memory to grow GDT).

Comments

The RPL of the selector returned from this call will be set to the DPL of the selector set in *DescDWORD1*.

The low 16 bits of the **EDX** return does not change, but it is safest to save the value of the GDT selector after each **Allocate_GDT_Selector** call. This selector will have DPL = RPL = 0, and the TI bit (bit 2) will be clear.

The high 16 bits of the **EDX** return must be saved after each call, if its value is important, because the size of the GDT may change on each call.

The prefered method of changing a GDT descriptor is to use **SetDescriptor**, rather than using the GDT selector which is returned by this call.

Allocate_LDT_Selector

```
unsigned long
Allocate_LDT_Selector(VMHandle,DescDWORD1,DescDWORD2,Count,flags)
unsigned VMHandle;
unsigned DescDWORD1;
unsigned DescDWORD2;
unsigned Count;
unsigned flags;
```

This call is used to create new LDT selector(s) in the specified VM context. *VMHandle* is a valid VM handle and indicates the VM context for which the selector(s) will be valid. *DescDWORD1* and *DescDWORD2* form the 8 bytes of information to be placed in the new descriptor(s). *DescDWORD1* is the high order 4 bytes of the descriptor containing the high 16 bits of the base, the high 4 bits of the limit and the status and type bits. *DescDWORD2* is the low order 4 bytes for the descriptor containing the low 16 bits of the base and limit. Use **BuildDescDWORDs** to help you set up these arguments. The *Count* parameter specifies the number of contiguous LDT selectors to allocate. This parameter supports Block Selector Assignment strategies. USE16 segmented applications cannot address objects larger than 64K Bytes in size without having multiple selectors that describe the sequential 64K Byte blocks of the object. For an object <=64K bytes in size, or instances where it is inappropriate, Count = 1. For an object >64K bytes in size, Count = (Size + (64K - 1))/64K. Notice that the selectors allocated for count >1 all have the same descriptor DWORDs in them. It is up to the caller to edit the base and limits of the individual selectors in a Block Selector Assignment using the LDT selector returned in the low 16 bits of **EDX**. There are currently no bits defined in the flags, this parameter must be set to 0.

Return Value

Returns a 64 bit long which is actually two 32 bit DWORDs. The low DWORD (**EAX**) is the nonzero selector if successful, if Count was >1, this is the FIRST selector, the second is **EAX**+8, the third **EAX**+16, etc. The high DWORD (**EDX**) is split into two 16 bit word returns. The low 16 bits of **EDX** is the LDT descriptor which describes the LDT itself. The allows the caller to do things such as change the present bit of LDT selectors and change the base and limit. The high 16 bits of **EDX** is the number of selectors currently in the LDT (the "limit" of the LDT expressed as a number of selectors, (LIMIT+1)/8). Both DWORDS have value 0 if the allocation failed (Bad DescDWORD arguments, LDT is full, invalid *VMHandle* insufficient memory to grow LDT).

Comments

The RPL of the selector returned from this call will be set to the DPL of the selector set in *DescDWORD1* and the TI bit (bit 2) will be set.

The high 16 bits of the **EAX** return are zero since selectors are 16 bit quantities.

Note that LDT selectors are PER VM and only valid in that VM context (VM must be current VM for selector to be valid). Use **SelectorMapFlat** to look at regions described by LDT selectors in VMs which are not the current VM.

The low 16 bits of the **EDX** return does not change once the LDT of a particular VM is created, but it is safest to save the value of the LDT selector after each **Allocate_LDT_Selector call.** This selector will have DPL = RPL = Protected Mode Application Privilege, and the TI bit (bit 2) will be set.

The high 16 bits of the **EDX** return *must be saved* after each call, if its value is important, because the size of the LDT may change on each call.

The multiple selectors allocated with Count >1 must be individually freed. **_Free_LDT_Selector** does not have a count.

The prefered method of changing an LDT descriptor is to use **SetDescriptor.**

Use of ALDTSpecSel is not advised. Reliance on specific "hard coded" LDT selectors is contrary to good system design principals. Note that a bit like this does not exist for **Allocate_GDT_Selector**, this is intentional. A call with this bit set may always fail for some values of the Count parameter, and it may start failing for all values of the Count parameter in a later release of the product.

BuildDescDWORDs

```
unsigned  long BuildDescDWORDs(DESCBase,DESCLimit,DESCType,DESCSize,flags)
unsigned  DESCBase;
```

```
unsigned  DESCLimit;
unsigned  DESCType;
unsigned  DESCSize;
unsigned  Flags
```

This call is used to help you build the *DescDWORD1* and *DescDWORD2* arguments for calls to **Allocate_LDT/GDT_Selector**. DESCBase is the 32 bit BASE for the descriptor. DESCLimit is the 20 bit LIMIT for the descriptor. *DESCType* specifies the type BYTE (Only low 8 bits of the parameter are valid, other bits must be 0) for the descriptor. This is the byte that occupies bits 8-15 of the high DWORD of the descriptor (Present bit, DPL and TYPE fields). *DESCSize* specifies bits 20-23 of the high DWORD of the descriptor (Granularity, Big/Default). Notice that these bits occupy bits 4-7 of the *DESCSize* parameter, other bits must be 0. In other words *DESCSize* specifies a byte just like DESCType where only the high 4 bits of the byte are specified.

Current flags bits:

```
BDDExplicitDPL  EQU     00000000000000000000000000000001B
```

All unused bits must be zero. BDDExplicitDPL, if set, indicates that the DPL value specified in the DESCType field is to be used. If this bit is clear, then the DPL specified in the DESCType field is ignored and the DPL returned will be set to the protected mode application RPL. Since most selectors are built for the use by protected mode applications, this provides a convienient way to build descriptors without having to actually know which ring protected mode applications run in.

Return Value Returns the low DWORD of the descriptor (*DescDWORD2*) in **EAX**, and the high DWORD of the descriptor (*DescDWORD1*) in **EDX**.

Comments If you are building selectors for use by Protected Mode applications use the built-in capability provided by not setting the BDDExplicitDPL bit. Do not make assumptions about which ring protected mode applications run in. The selection of a ring for PM applications will be changed in future revs of Windows.

Free_GDT_Selector

```
unsigned Free_GDT_Selector(Selector,flags)
unsigned Selector;
unsigned flags;
```

This call is used to free a GDT selector allocated with a previous **Allocate_GDT_Selector** call. *Selector* is the return from a previous **Allocate_GDT_Selector** call. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value Returns nonzero value if successful, returns zero if the free failed (invalid *Selector*).

Comments Certain system selectors cannot be freed since they are required for operation of WIN386.

Free_LDT_Selector

```
unsigned Free_LDT_Selector(VMHandle,Selector,flags)
unsigned VMHandle;
unsigned Selector;
unsigned flags;
```

This call is used to free a LDT selector allocated with a previous **Allocate_LDT_Selector** call. *VMHandle* indicates the VM context of the selector. *Selector* is the return from a previous **Allocate_LDT_Selector** call. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value Returns nonzero value if successful, returns zero if the free failed (invalid *Selector*, invalid *VMHandle*).

Comments The RPL bits of the passed Selector are ignored by this call.

GetDescriptor

```
unsigned long GetDescriptor(Selector,VMHandle,flags) unsigned Selector;
unsigned VMHandle;
unsigned flags;
```

This call is used to get a copy of the two descriptor DWORDs associated with the given LDT or GDT Selector. Selector is a GDT or LDT selector value to get the descriptor of. The *VMHandle* parameter is ignored if Selector is a GDT selector. If Selector is an LDT selector, then *VMHandle* indicates the appropriate VM context for the Selector. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value Returns the low DWORD of the descriptor (*DescDWORD2*) in **EAX**, and the high DWORD of the descriptor (*DescDWORD1*) in **EDX**. Returns zero in both DWORDs if there was an error (invalid selector, invalid VM handle).

Comments The high 16 bits of the Selector argument are ignored (this is because the 80386 CPU often sets them to somewhat random values when DWORD operations are performed on segment registers).

The RPL bits of Selector are ignored.

The *VMHandle* parameter must be valid for LDT selectors.

SetDescriptor

```
unsigned SetDescriptor(Selector,VMHandle,DescDWORD1,DescDWORD2,flags)
unsigned Selector;
unsigned VMHandle;
unsigned DescDWORD1;
unsigned DescDWORD2;
unsigned flags;
```

This call is used to set (change) the descriptor of the given Selector. Selector is a GDT or LDT selector value to set the descriptor of. The *VMHandle* parameter is ignored if Selector is a GDT selector. If Selector is an LDT selector, then VMHandle indicates the appropriate VM context for the Selector. *DescDWORD1* and *DescDWORD2* form the 8 bytes of information to be placed in the descriptor. *DescDWORD1* is the high ORDER 4 bytes of the descriptor containing the high 16 bits of the base, the high 4 bits of the limit and the status and type bits. *DescDWORD2* is the low ORDER 4 bytes for the descriptor containing the low 16 bits of the base and limit. Use **BuildDescriptorDWORDs** to help you set up these arguments. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value Returns non-zero value if succesfull, returns zero if it failed (invalid *Selector*, invalid *VMHandle*).

Comments The high 16 bits of the Selector argument are ignored (this is because the 80386 CPU often sets them to somewhat random values when DWORD operations are performed on segment registers).

The RPL bits of Selector are ignored.

The *VMHandle* parameter must be valid for LDT selectors.

19.4 System Heap Allocator

The purpose of the heap allocator is to provide a memory manager service to system components to allocate small (i.e., less than a page size) blocks of memory for long term or short term use.

NOTE All of these calls use the USE32 C calling convention. The true name of the procedure has an underscore in front (i.e., **HeapAllocate** is actually **_HeapAllocate**), and the arguments are pushed right to left (unlike the PL/M calling convention used by Windows, which is left to right). The return value(s) is returned in C standard **EDX:EAX**. It is the responsibility of the *caller* to clear the arguments off the stack. Registers **EAX**, **ECX**, and **EDX** are changed by calls. Registers **DS**, **ES**, **FS**, **GS**, **EBP**, **EDI**, **ESI**, and **EBX** are preserved.

The heap uses a boundary tag allocation scheme similar to the one used by the MS-DOS operating system. This has the benefit of not placing some fixed limit on the total number of heap blocks. It has the disadvantage of having a fixed overhead of extra space per block.

The heap overhead is about 16 bytes per block. Users should keep this in mind when allocating lots of objects of small size. Try to combine such needs into larger heap blocks to cut down on the overhead.

WARNING You are strongly warned against making assumptions about the placement and size of the heap boundary tag structures. Future versions of WIN386 may change this behavior of the heap.

NOTE 4 byte (DWORD) alignment is maintained on heap blocks. This could be increased in a later version, but at least DWORD alignment is guaranteed.

HeapAllocate

```
unsigned HeapAllocate(nbytes,flags)
   unsigned nbytes;
   unsigned flags;
```

This is the call to allocate a block from the heap. *nbytes* is a 32 bit unsigned integer which is the size, in bytes, of the block. Current flags bits:

```
HeapZeroInit    EQU     00000000000000000000000000000001B
```

All unused bits *must be zero*. HeapZeroInit, if set, indicates that if the allocation is succesful, the memory is to be initialized with value 0 in all bytes of the block. If HeapZeroInit is clear, the block will have completely random values in it.

Return Value The return value is the 32 bit RING 0 address (offset relative to standard WIN386 RING 0 DS) of the block. Value is 0 if the allocation failed (insufficient memory).

Comments Blocks are DWORD aligned as noted, but sizes do not have to be a multiple of 4.

There is no "protection" of the heap. Care must be taken not to overrun the size of your block. Failure to do this will result in odd behavior and crashes.

There is no "motion" of blocks in the heap (heap blocks are all fixed), except via **HeapReAllocate**, and therefore no compaction. You are advised not to use the heap in such a way as to severely fragment it. You will end up wasting lots of memory by doing this.

The Flag bit equates are defined by including VMM.INC, please use the equates.

Allocation of 0 length heap blocks is not allowed.

HeapFree

```
unsigned HeapFree(hAddress,flags)
   unsigned hAddress;
   unsigned flags;
```

This call is used to free an existing block of heap. *hAddress* is the value returned from a previous call to **HeapAllocate** or **HeapReAllocate** and indicates the block to be freed. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value Returns nonzero value if the block was succesfully freed, zero if the free was unsuccesful (invalid *hAddress*).

Comments None

HeapGetSize

```
unsigned HeapGetSize(hAddress,flags)
   unsigned hAddress;
   unsigned flags;
```

This call is used to get the size of an existing block of heap. *hAddress* is the value returned from a previous call to **HeapAllocate** or **HeapReAllocate** and indicates the block to get the size of. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value Returns the size, in bytes, of the block. Returns zero if there was an error (invalid *hAddress*).

Comments None

HeapReAllocate

```
unsigned HeapReAllocate(hAddress,nbytes,flags)
   unsigned hAddress;
   unsigned nbytes;
   unsigned flags;
```

This call is used to grow or shrink or reinitialize an existing block of heap. *hAddress* is the value returned from a previous **HeapAllocate** or **HeapReAllocate** call and indicates the block to be reallocated. *nbytes* is a 32 bit unsigned integer which is the new size in bytes of the block. Current flags bits:

```
HeapZeroInit        EQU     00000000000000000000000000000001B
HeapZeroReInit      EQU     00000000000000000000000000000010B
HeapNoCopy          EQU     00000000000000000000000000000100B
```

All unused bits *must be zero*. **HeapZeroInit**, if set, indicates that if the reallocation is succesful, and the reallocation is growing the size of the block, the "grow area" of the block is to be initialized with value 0 in all bytes. This bit is ignored on a reallocation which is not growing the size of the block. **HeapZeroReInit**, if set, indicates that the ENTIRE block is to be reinitialized with value zero in all bytes of the block. HeapNoCopy, if set, indicates that the previous contents of the block are irrelevant, and don't need to be copied into the newly sized block. There is no reason that more than one of these bits should be set. If none of the bits are set, the previous contents of the block are copied into the new block, up to the lesser of the size of the new block, and the size of the old block, and the "grow area", if any, is not initialized with anything.

Return Value

The return value is the 32 bit RING 0 address (offset relative to standard WIN386 RING 0 DS) of the new block. Value is 0 if the reallocation failed (insufficient memory, or invalid *hAddress*).

Comments

Do not make assumptions about the relationship between the passed in *hAddress* and the *hAddress* returned. Assume that the returned *hAddress* is always different than the passed in *hAddress*.

In the case where this call fails, the passed in *hAddress* block remains valid. In the case where this call works and returns a new *hAddress*, the passed in *hAddress* is no longer valid (old block has been HeapFreed).

There is no "protection" of the heap. Care must be taken not to overrun the size of your block. Failure to do this will result in odd behavior and crashes.

There is no "motion" of blocks in the heap (heap blocks are all fixed), and therefore no compaction. You are advised not to use the heap in such a way as to severely fragment it. You will end up wasting lots of memory by doing this.

Note that this call can be used to reset the contents of an existing heap block to 0 by setting nbytes to the current size of the block and setting **HeapZeroReInit**.

You cannot **HeapReAllocate** a block to size 0, use **HeapFree**.

The Flag bit equates are defined by including VMM.INC, please use the equates.

19.5 System Page Allocator

The purpose of the page allocator is to provide the main allocation of 80386 4K pages to particular VM or VxDs.

NOTE All of these calls use the USE32 C calling convention. The true name of the procedure has an underscore in front (i.e., **PageAllocate** is actually **_PageAllocate**), and the arguments are pushed right to left (unlike the PL/M calling convention used by Windows, which is left to right). The return value(s) is returned in C standard **EDX:EAX**. It is the responsibility of the *caller* to clear the arguments off the stack. Registers **EAX**, **ECX**, and **EDX** are changed by calls. Registers **DS**, **ES**, **EBP**, **EDI**, **ESI**, and **EBX** are preserved.

CopyPageTable

```
unsigned CopyPageTable(LinPgNum,nPages,PageBufPTR,flags)
   unsigned LinPgNum;
   unsigned nPages;
   unsigned *PageBufPTR;
   unsigned flags;
```

This call is used to obtain a copy of a WIN386 page table. This call is intended as an assist to WIN386 system components that need to analyze the linear to physical mapping (such as DMA devices). *LinPgNum* is the page number of the first page of the range. This can be anything in the range 0 - 0FFFFFh. Thus addresses in the range 0-3FFh refer to addresses in the 1M V86 address space of the current VM. To compute the page number of any region simply take the address relative to the standard RING 0 WIN386 DS and shift it right by 12 bits. For example, the linear address 60001AB6h is in page number 60001h. Alignment considerations of this address (beyond 4K alignment) are the responsibility of the caller. *nPages* is the number of page table entries to copy. *PageBufPTR* is a 32 bit RING 0 offset relative to the standard WIN386 RING 0 DS which is the address of a buffer where the page table will be copied. Caller must insure that this buffer is large enough. Each page table entry is a DWORD, so the buffer must be at least nPages*4 bytes long. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value

Returns a nonzero value if the copy is succesful, returns 0 value if the copy was succesful, *but* at least a part of the range overlapped a region where the corresponding Page Directory Entry is not present.

Comments

You get a copy of the Page Table; writing to your buffer has no effect.

Note that V86 page tables stop at page 10Fh.

To look at the page table of a VM that is not the current VM simply use the high linear address of the VM. For instance to look at the page table starting at V86 address 0A000:0 of a VM which is not the current VM go:

```
mov eax,0A0000h                  ; V86 linear adress of 0A000:0
add eax,[ebx.CB_High_Linear]     ; High linear address
shr eax,12                       ; Convert to page number
```

Note that the above sequence always works correctly (works if the VM is the current VM as well). So simply doing this in all cases avoids the complication of worrying about whether the VM is the current VM.

The intent of this call is for you to look at the physical addresses in the high 20 bits of the entries. The low 12 bits of system information may be examined however.

You are warned to be careful about keeping this buffer for any length of time. The actual page table entries can change while the copy you got won't. The information in the copy should be analysed quickly.

GetDemandPageInfo

```
void GetDemandPageInfo(BufPtr,flags)
   DemandInfoStruc *BufPtr;
   unsigned flags;
```

This call is for use by the demand paging device. It provides information for the demand pager.

```
DemandInfoStruc struc
  DILin_Total_Count         dd     ?            ; Size of linear address space in pages
  DIPhys_Count              dd     ?            ; Count of phys pages
  DIFree_Count              dd     ?            ; Count of free phys pages
  DIUnlock_Count            dd     ?            ; Count of unlocked phys Pages
  DILinear_Base_Addr        dd     ?            ; Base of pageable address space
  DILin_Total_Free          dd     ?            ; Total free linear pages
  DIReserved                dd     12 DUP(?) ; Reserved
DemandInfoStruc ends
```

DILin_Total_Count is the size in pages of the linear address space subject to demand paging. DILinear_Base_Addr is the linear address of the start of the demand pageable region. Thus there are DILin_Total_Count pages starting at address DILinear_Base_Addr which are subject to demand paging. DILin_Total_Free is the number of the DILin_Total_Count pages which are currently free. Notice that this space may not be allocatable in a single block, it is the total free, not the size of the largest free block. Note that if DILinear_Base_Addr == 0, this means that the demand pageable region of the system is not contiguous. DIPhys_Count is the total number of physical pages under the control of the memory manager. DIFree_Count is the number of pages currently on the free list. DIUnlock_Count is the count of pages which are currently unlocked, notice that free pages are unlocked. There are currently no bits defined in the flags, this parameter must be set to 0.

Return Value This call does not have a return value. It simply fills in the structure pointed to by *BufPtr*.

Comments The reserved field is exactly that, reserved. Do not make any assumptions about what is in this region. Behavior will change in later releases.

GetFreePageCount

```
unsigned long (flags)
   unsigned flags;
```

This call is used to obtain the count of free 4K pages. And the count of pages that can be allocated as PageLocked. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value The return value is a 64 bit long which is actually two 32 bit DWORDS. The Low DWORD (**EAX**) is the 32 bit count of free 4K pages in the system which could be allocated with the **PageAllocate** call. The High DWORD (**EDX**) is the 32 bit count of pages available for allocation as PageLocked pages at the current time.

Comments You should be careful about making assumptions about being able to turn around and issue a call to allocate all of the pages returned by this call. Besides any alignment considerations, it is possible someone could get in and allocate some or all of the pages before you. This call is intended to be advisory in nature.

Note that in a demand paged virtual memory system such as WIN386 the free pages count is usually very close to 0. It is more relevant to use the **EDX** return to make judgements about allocation possibility. **EDX** contains the count of pages currently available for allocation as **PageLocked** pages. Note that many assumptions are not valid. **EAX<=EDX** is *not* a valid assumption for instance.

Note that in a virtual memory environment it is not a good idea to go soaking up tons of virtual address space. Start with some, then **PageReAllocate** it to make it bigger if needed.

GetSetPageOutCount

```
unsigned GetSetPageOutCount(NewCount,flags)
   unsigned NewCount;
   unsigned flags;
```

This call is for use by the demand paging device. It allows the paging device to manipulate a memory manager parameter associated with demand paging. This parameter is the "page out ahead" count. Whenever a page is paged out to satisfy a page in, an additional PageOutCount-1 pages are also paged out and put on the free list (if possible).There is one bit in the flags:

```
GSPOC_F_Get  equu   00000000000000000000000000000001B
```

All other bits must be zero. If GSPOC_F_Get is set, the call returns the current value of the page out count in **EAX**, and the *NewCount* parameter is ignored. If GSPOC_F_Get is not set, the call sets the value of the page out count to *NewCount*.

Return Value Returns the page out count if GSPOC_F_Get is set, else it has no return.

WARNING This call is intended for use by the PageSwap device, others should not be calling it! Others making this call can disturb the operation of the PageSwap device.

Comments There is an equate for the flag bit in VMM.INC, use the equate.

GetSysPageCount

```
unsigned GetSysPageCount(flags)
   unsigned flags;
```

This call is used to obtain the current count of system (PG_SYS) 4K pages. There are currently no bits defined in the *flags*, and this parameter must be set to 0.

Return Value The return value is the 32 bit count of 4K pages allocated as PG_SYS pages in the system.

Comments It is generally true that this number is the size of WIN386. However, this is the general case only.

GetVMPgCount

```
unsigned long GetVMPgCount(VMHandle,flags)
   unsigned VMHandle;
   unsigned flags;
```

This call is used to get the current count of 4K pages allocated to a particular VM. The *VMHandle* parameter must be a valid VM handle and indicates the VM to get the allocated page count of. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value The return value is a 64 bit long which is actually two 32 bit DWORDS. The Low DWORD (**EAX**) is the total count of pages (of all types but PG_SYS) in the system allocated for this VM. The High DWORD (**EDX**) is the count of pages which are allocated to this VM, but which are not mapped into the VM's 1Meg address space at the current time. Value (both dwords) is 0 if the call failed (invalid *VMHandle*).

Comments You should be careful about assuming that **EAX-EDX** is the size of the VM. It is in one sense, but not in the standard DOS senses.

MapIntoV86

```
unsigned MapIntoV86(hMem,VMHandle,VMLinPgNum,nPages,PageOff,flags)
   unsigned hMem;
   unsigned VMHandle;
   unsigned VMLinPgNum;
   unsigned nPages;
   unsigned PageOff;
   unsigned flags;
```

This call is used to map some or all of the pages of a memory block into a specific VM's Virtual 8086 address space. *hMem* is the value returned from a previous call to **PageAllocate** or **PageReAllocate** and indicates the block to be mapped. *VMHandle* parameter must be a valid VM handle and indicates the VM into which the map is to occur. *VMLinPgNum* is the address in the 1M V86 address space where the map will start (this is a page number, thus linear address 60000h = page 60h). Alignment considerations of this address (beyond 4K alignment) are the responsibility of the caller. Map addresses below page 10h, or above 10Fh will cause an error. *nPages* is the number of pages to map. *PageOff* is the number of pages into the *hMem* block to the first page of the block which is to be mapped at *VMLinPgNum* (thus PageOff is 0 to map the first page of *hMem* at VMLinPgNum). *nPages* and *PageOff* allow one *hMem* block to be scatter mapped into different VM locations. An error will occur if *PageOff* + *nPages* is greater than the size of *hMem*.

Current flags bits:

```
PageDEBUGNulFault       EQU     00000000000000000000000000010000B
```

All unused bits must be zero. PageDEBUGNulFault, if set, indicates that if *hMem* is the handle of the NUL system page, and this is the DEBUG version of WIN386, access to these pages should cause a page fault DEBUG exception. This bit is ignored if *hMem* is not the system NUL page handle, or this is not DEBUG WIN386.

It is generally true that *hMem* blocks mapped with this call should not be composed of PG_SYS pages. This is not disallowed, but is not advised.

There is a special *hMem* handle that can be used with this call. The value of this handle is obtained by calling the routine **GetNulPageHandle** (actual name **_GetNulPageHandle**) which will return you this special *hMem* handle in **EAX**. This is the *hMem* of the system NUL page. This page is used to occupy regions of the address space which are "unused" but for which it is not desirable to cause a page fault if they are accessed. The NUL page is multiply mapped at many locations in the system, so its contents are always random. Under DEBUG WIN386, a fault occurs if the NUL page is touched and the PageDEBUGNulFault bit was set on the call which mapped the page.

If the **PageSwap** device is type one (*not* direct to hardware), there is an *implied* PageLock on the pages mapped with this call, and an *implied* **PageUnlock** on the pages which this call is mapping over. This is consistent with the fact that pages mapped into V86 address space must be locked (V86 memory cannot be demand paged). If the **PageSwap** device is type two (direct to hardware) than the implied lock and unlock done by this call are disabled because in the case of a type two **PageSwap** device V86 memory CAN be demand paged. See the **PageAllocate** documentation for a description of the different **PageSwap** device types and their relevance.

Return Value

Returns a nonzero value if the map is succesful, returns 0 value if the map was unsuccesful (invalid *hMem*, invalid *VMHandle*, map range illegal, size discrepancy, insufficient memory on implied PageLock).

Comments

The implied **PageLock**, which is performed on all of the pages mapped if the **PageSwap** device is type oneAg, is consistent with the fact that V86 memory cannot be Demand Paged while the VM is in a runable state. Whenever the V86 memory mapping is changed via **MapIntoV86**, the previous memory that was mapped in that region of the VM is unlocked. The correct way to think of this is that there is an implied **PageLock** whenever memory is mapped into a V86 context, and an implied **PageUnlock** whenever it is "unmapped" from the V86 context. This "unmapping" can occur when: A different handle (including the NulPageHandle) is **MapIntoV86ed** or **LinMapIntoV86ed** to the region, or a **PhysIntoV86** is performed to the region.

There is nothing to prevent you from mapping the same block, or piece of a block, into multiple places in a VM, or into multiple VMs. Such operations are not particularly advisable though. For one thing, the reporting of memory owned by a VM will be disturbed. For this reason it is also not generally a good idea to map pages that were allocated as belonging to one VM into a different VM. The one exception to this general rule is the request for a map by one VM to look at the memory of a different VM. Such maps should be of a relatively short duration.

The page attributes for these pages will be P_USER+P_PRES+P_WRITE. P_DIRTY and P_ACC will be cleared by the call. PG_TYPE will be set to whatever the type of the hMem pages are.

The Flag bit equates are defined by including VMM.INC, please use the equates.

The intent of **MapIntoV86** support for pages between page 10h and FirstV86Page is to support WIN386 devices which have **Allocate_Global_V86_Data_Area** a **GVDAPageAlign** region. Use of mapping in this region to other addresses can easily crash the system and should be avoided.

Regions which span across **FirstV86Page** are not allowed.

The reason for the page 10h limitation is that on most versions of the Intel 80386 CPU there is an errata which prevents you from setting up a Linear != Physical address mapping in the first 64K of the address space.

ModifyPageBits

```
unsigned ModifyPageBits(VMHandle,VMLinPgNum,nPages,bitAND,bitOR,pType,flags)
   unsigned VMHandle;
   unsigned VMLinPgNum;
   unsigned nPages;
```

```
unsigned bitAND;
unsigned bitOR;
unsigned pType;
unsigned flags;
```

This call is used to modify the page protection bits associated with PG_HOOKED pages in the V86 address space of a VM. It allows the P_PRES, P_WRITE, and P_USER bits of the pages to be modified along with PG_TYPE if appropriate. The *VMHandle* parameter must be a valid VM Handle and indicates the VM whose page bits are to be modified. *VMLinPgNum* is the page number in the 1M V86 VM address space where the modification will start (this is a page number, thus linear address A0000h = page A0h). When clearing the P_PRES bit (making pages not present), all of the pages specified (*nPages* starting at *VMLinPgNum*) must be PG_HOOKED pages for which a HOOK Page Fault handler has been registered, and *pType* must be PG_HOOKED. *nPages* is the number of pages to modify the bits of. Addresses below the start of VM specific memory, or above 10Fh will cause an error. *bitAND* is an AND mask for the bits, *bitOR* is an OR mask. Thus to clear P_PRES, P_WRITE, and P_USER, *bitAND* would be (*not* P_PRES+P_WRITE+P_USER), and *bitOR* would be zero. To set P_USER, and clear P_WRITE, leaving P_PRES unchanged, bitAND would be (NOT P_WRITE), and *bitOR* would be P_USER. Having bits other than P_WRITE, and P_USER set in *bitOR* will cause an error. Having bits other than P_PRES, P_WRITE, and P_USER clear in *bitAND* will cause an error.

This call always has the side effect of clearing P_DIRTY and P_ACC. Thus to just clear these two bits, give a *bitAND* of 0FFFFFFFFh, and a *bitOR* of 0. *pType* indicates a value to be placed in the PG_TYPE field. The allowed values are:

```
PG_HOOKED   EQU 7
PG_IGNORE   EQU -1      (0FFFFFFFFh)
```

Any other value will cause an error. PG_IGNORE indicates that the PG_TYPE field is not to be modified by the call. This is the value that *must* be set if P_PRES bit is being set (or being left set). PG_HOOKED must be specified if the P_PRES bit is being cleared by the call. Recall that making a PhysIntoVM call sets the type field for the physical pages to PG_SYS. This parameter is provided so that the page types can be reset to PG_HOOKED when the mapping is changed to not present. Recall that MapIntoVM also resets the PG_TYPE field to the type of the pages of *hMem*. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value

Returns a nonzero value if successful, returns 0 value if unsuccessful (invalid *VMHandle*, invalid bits in *bitAND* or *bitOR*, invalid *pType*, page range bad).

Comments

You cannot use this call to set the Present bit. You may either clear the present bit, or leave it unaffected. Use **MapIntoV86** or **PhysIntoV86** to make pages present.

PageAllocate

```
unsigned PageAllocate(nPages,pType,VMHandle,AlignMask,minPhys,
                        maxPhys,PhysAddrPTR,flags)
        unsigned nPages;
        unsigned pType;
        unsigned VMHandle;
        unsigned AlignMask;
        unsigned minPhys;
        unsigned maxPhys;
        unsigned *PhysAddrPTR;
        unsigned flags;
```

This is the call to allocate a block of memory. The memory allocated is actually just linear address space, whether there is actually physical memory mapped for this block as part of the allocation is specified by the flags. *nPages* is a 32 bit unsigned integer which is the size in 4K pages of the block. *pType* indicates the type of page(s) being allocated:

```
PG_VM           EQU             0
PG_SYS          EQU             1
PG_HOOKED       EQU             7
```

PG_VM pages are pages which are specific to a particular VM context. The handle of PG_VM memory blocks will typically be placed in the VM Control Block someplace. PG_HOOKED pages are pages which will be mapped into the VM at locations where the component has registered a HookPageFault handler. Like PG_VM pages, PG_HOOKED pages are specific to a particular VM context. The *VMHandle* parameter must be a valid VM Handle for all page types except PG_SYS. PG_SYS pages are global system pages which are valid in all VM contexts (pages are specific to the WIN386 system component which allocates them, rather than to a VM). The *VMHandle* parameter is not relevant to PG_SYS pages and it *must* be set to 0 when allocating PG_SYS pages.

Current flags bits:

```
PageZeroInit        EQU     00000000000000000000000000000001B
PageUseAlign        EQU     00000000000000000000000000000010B
PageContig          EQU     00000000000000000000000000000100B
PageFixed           EQU     00000000000000000000000000001000B
PageLocked          EQU     00000000000000000000000010000000B
PageLockedIfDP      EQU     00000000000000000000000100000000B
```

All unused bits *must be zero*. **PageLocked**, if set, indicates that a **PageLock** is implied as part of the **PageAllocate** operation. This forces the allocate to make all pages of the handle present when the handle is allocated consistent with the implied **PageLock**. **PageLockedIfDP**, if set, indicates that a **PageLock** is implied as part of the **PageAllocate** *only if the* **PageSwap** *device is not direct to hardware*. There are two basic behavior types for the **PageSwap** device. Type one pages through DOS and/or the ROM BIOS. This type of **PageSwap** device places restrictions on the ability to demand page certain types of system memory because of the fact that it runs partly in V86 mode as part of its operation. **PageSwap** type two pages by talking directly to the disk hardware. This second type of **PageSwap** device removes some of the restrictions because it runs completely in protected mode when accessing the paging device. **PageLocked** indicates that the memory should be locked regardless of which type of **PageSwap** device is present. **PageLockedIfDP** indicates that this memory only needs to be locked if the **PageSwap** device is type one. PageFixed, if set, indicates behavior similar to **PageLocked** as far as the implied **PageLock** is concerned, and in addition a Fixed handle can *never* be unlocked, and its linear address will *never* change (via **PageReAllocate**). Note that ReAllocation of a Fixed handle will generally not succeed due to the Fixed restriction on the ability to change the linear address of the handle. Note that an allocation without an implied **PageLock** via **PageLocked**, **PageLockedIfDP**, or **PageFixed** will simply allocate linear address space. The pages of such a handle will be made present "on demand" when the address space is touched. If it is desired to make part of the handle present to perform some function, use **PageLock** to force the contents to be loaded. PageUseAlign, if set, indicates that the *AlignMask*, *minPhys*, *maxPhys*, and *PhysAddrPTR* parameters are specified. If PageUseAlign is clear, the *AlignMask*, *minPhys*, *maxPhys*, and *PhysAddrPTR* parameters are set to 0 and ignored. Note that if PageUseAlign is set, PageFixed must also be specified. It makes no sense to have an aligned memory handle which is not fixed. PageZeroInit, if set, indicates that if the allocation is succesful, the memory is to be initialized with value 0 in all bytes of the block. If PageZeroInit is clear, the block will have completely random values in it. PageContig, if set, indicates that the Physical memory pages of the block are to occupy sequential Physical memory addresses (memory is "physically contiguous"). *PageContig is ignored if PageUseAlign is not set.*

PageUseAlign is provided to assist device drivers that wish to allocate buffers for use by the device which have additional alignment restrictions enforced by the hardware (such as 64K and 128K alignment for DMA). If the PageUseAlign bit is set, AlignMask specifies an alignment (power of 2> 4k) requirement for the first physical page of the block. Physical page numbers are the physical address of the page shifted right by 12. Correct alignment is tested for by ANDing *AlignMask* with the first physical page number and testing for zero. If the AND is zero, the page has the correct alignment. Thus:

```
00000000h    =      4K alignment (ignore AlignMask)
00000001h    =      8K alignment
00000003h    =     16K alignment
00000007h    =     32K alignment
0000000Fh    =     64K alignment
0000001Fh    =    128K alignment
```

Remember that you will probably also want to set the PageContig bit. *minPhys* and *maxPhys* place additional physical address restrictions on the physical pages of the memory block. These specify the minimum and maximum allowed physical page numbers. All physical page numbers of the block must be >=minPhys, and <maxPhys. For instance, for setting up a DMA buffer for an 80386 accelerator card in a PC XT, the buffer needs to be physically restricted to pages less than 1 MB since the XT DMA controller cannot DMA into pages above 1 MB. In this case, minPhys would be 0, and maxPhys would be 100h. If you don't want to specify this (i.e. you just want *AlignMask*), set *minPhys* to 0, and maxPhys to 0FFFFFFFFh. Note that when PageUseAlign is set, the physical page address (physical page number shifted left by 12) of the start of the block will be returned via the *PhysAddrPTR* pointer parameter.

NOTE PageUseAlign PageAllocations can only be performed during device initialization. Aligned PageAllocations will fail if done after device initialization.

Return Value

The return value is a 64 bit long which is actually two 32 bit DWORDS. The Low DWORD (**EAX**) is the memory handle of the block. The High DWORD (**EDX**) is the 32 bit RING 0 address (offset relative to standard WIN386 Ring 0 DS) of the block. If **PageUseAlign** was specified, the physical address of the start of the block is placed in the DWORD pointed to by **PhysAddrPTR**. Value (both DWORDs) is 0 if the allocation failed (insufficient memory).

Comments

You should be careful about making assumptions about any apparent relationship between the memory handle and the blocks RING 0 or physical address. Any such apparent relationship is subject to change in a later release.

PhysAddrPTR had better point somewhere reasonable when PageUseAlign is specified. There is no way to check its validity, if it's garbage you'll either cause a page fault or stomp on something you shouldn't.

PageAllocation of 0 length blocks is not allowed.

PageLocked and **PageLockedIfDP** should not both be set. Only one, or the other, or neither are valid settings. *Note also that* **PageLockedIfDP** *cannot be set on calls made before the init complete system control call is made*. This is because it is not possible to ask the **PageSwap** device what type it is before it has been initialized.

The Flag bit equates are defined by including VMM.INC, please use the equates.

PageFree

```
unsigned PageFree(hMem,flags)
   unsigned hMem;
   unsigned flags;
```

This call is used to free an existing block of pages. *hMem* is the value returned from a previous call to **PageAllocate** or **PageReAllocate** and indicates the block to be freed. There are currently no bits defined in the flags, this parameter must be set to 0.

Return Value

Returns nonzero value if the block was succesfully freed, zero if the free was unsuccesful (invalid *hMem*).

Comments

It is the responsibility of the WIN386 system components which allocate non-PG_SYS pages to free them when the VM they are associated with is destroyed. There is no "automatic" freeing of such memory done by the memory manager. PG_SYS pages do not need to be freed before WIN386 exits.

It is not an error to **PageFree** a handle which is all or partially locked.

WARNING Be very careful about PageFreeing blocks which are currently MapIntoV86ed to some VM context. Doing this can result in a crash.

PageGetAllocInfo

```
unsigned long PageGetAllocInfo(flags)
 unsigned flags;
```

This call is used to obtain information prior to a **PageAllocate** or **PageReallocate** call. It returns the largest block of linear address space that could be allocated, together with information relating to allocation of Locked or Fixed memory. There are currently no bits defined in the flags, this parameter must be set to 0.

Return Value The return value is a 64 bit long which is actually two 32 bit DWORDs. The Low DWORD (**EAX**) is the 32 bit count of free 4K pages in the system which could be allocated with the **PageAllocate** as *not* **PageLocked** or **PageFixed** memory. The High DWORD (**EDX**) is the 32 bit count of pages available for allocation as **PageLocked** pages at the current time.

Comments You should be careful about making assumptions about being able to turn around and issue a call to allocate all of the pages returned by this call. Besides any alignment considerations, it is possible someone could get in and allocate some or all of the pages before you. This call is intended to be advisory in nature.

EAX contains the size of the largest available region of linear address space. **EDX** contains the count of pages currently available for allocation as **PageLocked** pages. Notice that many assumptions are not valid. **EAX >= EDX** is *not* a valid assumption for instance.

You should be very careful about turning around and doing a **PageAllocate** with the **EAX** return from this call. You can cause all sorts of odd behavior if you take up all of the linear address space. You should allocate memory on an as needed basis instead of allocating huge blocks of memory most of which you do not use.

PageGetSizeAddr

```
unsigned long PageGetSizeAddr(hMem,flags)
   unsigned hMem;
   unsigned flags;
```

This call is used to get the size and linear address of an existing block of pages. *hMem* is the value returned from a previous call to **PageAllocate** or **PageReAllocate** and indicates the block to get the size and address of. There are currently no bits defined in the flags, this parameter must be set to 0.

Return Value

The return value is a 64 bit long which is actually two 32 bit DWORDS. The Low DWORD (**EAX**) is the size in 4K pages of the block. The High DWORD (**EDX**) is the 32 bit RING 0 address (offset relative to standard WIN386 Ring 0 DS) of the block. Value (both DWORDs) is 0 if the call failed (invalid *hMem*).

Comments

Note that the size of a handle is the total size of the handle and has nothing to do with what pieces of the handle may or may not be present.

PageLock

```
unsigned PageLock(hMem,nPages,PageOff,flags)
   unsigned hMem;
   unsigned nPages;
   unsigned PageOff;
   unsigned flags;
```

This call is used to lock (make present) all or part of an existing memory handle. *hMem* is the value returned from a previous call to **PageAllocate** or **PageReAllocate** and indicates the block to be locked. *nPages* specifies the count of pages to be locked. *PageOff* specifies the page offset from the start of the block of the first page to be locked. *nPages* together with *PageOff* allow all or only part of the *hMem* block to be locked. An error will occur if *PageOff+nPages* is greater than the size of *hMem*. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Current flags bits:

```
PageLockedIfDP  EQU     00000000000000000000001000000000B
```

All unused bits *must be zero*. **PageLockedIfDP**, if set, indicates that the lock only needs to be done if the **PageSwap** device is not direct to hardware. In the case where the **PageSwap** device is of type two (direct to hardware), calls to this routine with **PageLockedIfDP** set are effectively NOPs. See the **PageAllocate** documentation for a description of the different **PageSwap** device types and their relevance.

Return Value

Returns nonzero value if the block was succesfully locked, zero if the lock was unsuccesful (invalid *hMem*, insufficient memory).

Comments

This call may be issued on *hMem* blocks which are **PageFixed**, but this is a wasted call since **PageFixed** blocks are always locked (present).

Because of the overcommit associated with demand paging, callers must be prepared for this call to fail due to unavailability of sufficient memory to make the region present.

Note that **PageLockedIfDP** *cannot be set on calls made before the init complete system control call is made*. This is because it is not possible to ask the **PageSwap** device what type it is before it has been initialized.

Each Page of a handle has an individual lock count. Each lock increments the counter. The counter must go to 0 for the page to be unlocked. This means that if the handle is locked 5 times, it has to be unlocked 5 times.

Do not leave handles locked when they don't need to be, unlock handles as soon as possible to make the physical memory associated available for use by demand paging.

The Flag bit equates are defined by including VMM.INC, please use the equates.

PageOutDirtyPages

```
unsigned PageOutDirtyPages(nPages,flags)
   unsigned nPages;
   unsigned flags;
```

This call is for use by the demand paging device. It allows the paging device to periodically "flush" out dirty pages to prevent a large number of dirty pages from accumulating in the system. *nPages* is the maximum number of dirty pages to flush at this time.

Current flags bits:

```
PagePDPSetBase        EQU      00000000000000000100000000000000B
PagePDPClearBase      EQU      00000000000000001000000000000000B
PagePDPQueryDirty     EQU      00000000000000100000000000000000B
```

All unused bits *must be zero*. The **PageSwap** device may wish to flush out all dirty pages in the system as part of a "background" activity ("write out ahaead"). These two bits allow this to be done, it allows the caller to manipulate a variable associated with the page out scan which will cause the scan to stop. This "base" page number that is set allows the **PageSwap** device to tell when the **PageOutDirtyPages** call has completed a scan of the entire address space looking for dirty pages. **PagePDPSetBase** tells **PageOutDirtyPages** to set the base page number to the current scan start point. **PagePDPClearBase** tells **PageOutDirtyPages** to clear the base page number, setting it to NONE. A return value of 0 is used to detect when a **PageOutDirtyPages** call has stoped because it has hit the base page. This is not totally reliable, but is a reasonable approximation, since **PageOutDirty-Pages** can return 0 because there are no dirty pages (this is rather unlikely though). **PagePDPQueryDirty**, if set, indicates that the call is to return the current count of DIRTY demand pageable pages, the *nPages* argument and all other flags are ignored if this bit is set (call returns the count of dirty pages as its sole function).

Return Value Returns the actual count of dirty pages flushed by the call (0 is valid).

WARNING This call is intended for use by the PageSwap device, others should not be calling it! Others making this call can disturb the operation of the PageSwap device.

Notes This call functions something like a partial "commit" of the dirty pages in the system. Note that ALL of the dirty demand pages can be flushed by specifying a large value for *nPages* (like 0FFFFFFFFh).

This call operates only on current page out candidates.

The Flag bit equates are defined by including VMM.INC, please use the equates.

PageReAllocate

```
unsigned PageReAllocate(hMem,nPages,flags)
   unsigned hMem;
   unsigned nPages;
   unsigned flags;
```

This call is used to grow or shrink or reinitialize an existing block of memory. *hMem* is the value returned from a previous **PageAllocate** or **PageReAllocate** call and indicates the block to be reallocated. *Note that handles allocated with* **PageUseAlign** *set cannot be* **PageReAllocated.** *nPages* is a 32 bit unsigned integer which is the new size in 4K pages of the block.

Current flags bits:

```
PageZeroInit        EQU    00000000000000000000000000000001B
PageZeroReInit      EQU    00000000000000000000000000100000B
PageNoCopy          EQU    00000000000000000000000001000000B
PageLocked          EQU    00000000000000000000000010000000B
PageLockedIfDP      EQU    00000000000000000000000100000000B
```

All unused bits *must be zero*. **PageLocked** and **PageLockedIfDP**, if set, indicates that if this **PageReAllocation** is growing the size of the handle, the pages added to the handle are to be **PageLocked** or **PageLockedIfDP** (see **PageAllocate** for explanation). If the **PageReAllocation** is not growing the handle these bits are ignored. *Note that* **PageFixed** *is not specified*, **PageReAllocation** of a **PageFixed** handle is implied as **PageFixed** by the handle itself. **PageZeroInit**, if set, indicates that if the reallocation is succesful, and the re-allocation is growing the size of the block, the "grow area" of the block is to be initialized with value 0 in all bytes. This bit is ignored on a re-allocation which is not growing the size of the block. **PageZeroReInit** , if set, indicates that the entire block is to be reinitialized with value zero in all bytes of the block. **PageNoCopy**, if set, indicates that the previous contents of the block are irrelevant, and don't need to be copied into the newly sized block. There is no reason that more than one of these three bits should be set. If none of the bits are set, the previous contents of the block are copied into the new block, up to the lesser of the size of the new block, and the size of the old block, and the "grow area", if any, is not initialized with anything.

Return Value

The return value is a 64 bit long which is actually two 32 bit DWORDS. The Low DWORD (**EAX**) is the memory handle of the new block. The High DWORD (**EDX**) is the 32 bit RING 0 address (offset relative to standard WIN386 Ring 0 DS) of the block. Value (both DWORDs) is 0 if the reallocation failed (insufficient memory, handle wrong type, invalid handle).

Comments

Do not make assumptions about the relationship between the passed in *hMem* and the Address returned, if specified. Assume that the returned *hMem* and address are always different than the passed in *hMem* and previous address.

In the case where this call fails, the passed in *hMem* and previous address of the block remain valid. In the case where this call works and returns a new *hMem* and address, the passed in *hMem* and previous address are no longer valid (old block has been PageFreed).

WARNING Be very careful about PageReAllocating blocks which are currently MapIntoV86ed to some VM context. Doing this can result in a crash.

PageLocked and **PageLockedIfDP** should not both be set. Only one, or the other, or neither are valid settings. *Note also that* **PageLockedIfDP** *cannot be set on calls made before the init complete system control call is made*. This is because it is not possible to ask the **PageSwap** device what type it is before it has been initialized.

Note that this call can be used to reset the contents of an existing block to 0 by setting *nPages* to the current size of the block and setting PageZeroReInit.

You cannot **PageReAllocate** a block to size 0, use **PageFree**.

The Flag bit equates are defined by including VMM.INC, please use the equates.

PageUnLock

```
unsigned PageUnLock(hMem,nPages,PageOff,flags)
   unsigned hMem;
   unsigned nPages;
   unsigned PageOff;
   unsigned flags;
```

This call is used to unlock all or part of an existing memory handle that was previously locked. *hMem* is the value returned from a previous call to **PageAllocate** or **PageReAllocate** and indicates the block to be unlocked. *nPages* specifies the count of pages to be unlocked. *PageOff* specifies the page offset from the start of the block of the first page to be unlocked. *nPages* together with *PageOff* allow all or only part of the *hMem* block to be unlocked. An error will occur if *PageOff+nPages* is greater than the size of *hMem*.

Current flags bits:

```
PageLockedIfDP          EQU     00000000000000000000000100000000B
PageMarkPageOut         EQU     00000000000000000010000000000000B
```

All unused bits must be zero. **PageLockedIfDP**, if set, indicates that the unlock only needs to be done if the **PageSwap** device is not direct to hardware. In the case where the **PageSwap** device is of type two (direct to hardware), calls to this routine with **PageLockedIfDP** set are effectively NOPs. See the **PageAllocate** documentation for a description of the different **PageSwap** device types and their relevance. **PageMarkPageOut**, if set, indicates that if this unlock actually does unlock the pages (lock count goes to 0) the pages are to be made prime candidates for page out. This flag should only be set if it is unlikely that these pages are going to be touched for a while. Effectively what this does is clear the P_ACC bits of the pages which causes them to be first level page out candidates.

Return Value

Returns nonzero value if the block was succesfully unlocked, zero if the lock was unsuccesful (invalid *hMem*, no part of range is locked).

Comments

This call *may* be issued on *hMem* blocks which are **PageFixed**, but this is a wasted call since **PageFixed** blocks cannot be unlocked.

Note that **PageLockedIfDP** *cannot be set on calls made before the init complete system control call is made.* This is because it is not possible to ask the **PageSwap** device what type it is before it has been initialized.

Each page of a handle has an individual lock count. Each lock increments the counter. The counter must go to 0 for the page to be unlocked. This means that if the handle is locked 5 times, it has to be unlocked 5 times.

The Flag bit equates are defined by including VMM.INC, please use the equates.

PhysIntoV86

```
unsigned PhysIntoV86(PhysPage,VMHandle,VMLinPgNum,nPages,flags)
   unsigned PhysPage;
   unsigned VMHandle;
   unsigned VMLinPgNum;
   unsigned nPages;
   unsigned flags;
```

This call is very similar to the **MapIntoV86** call only instead of taking a memory handle argument, it takes a Physical address (page number). The intent of this call is to "hook up" a particular VM to the actual Physical device memory of a device (such as the video memory of a display adaptor). *PhysPage* is the physical page number of the start of the region to be mapped, and indicates the block of physical memory to be mapped. For instance, to hook up to the 64K of video memory at A000:000, PhysPage would be A0h and *nPages* would be 10h. The *VMHandle* parameter must be a valid *VM handle* and indicates the VM into which the map is to occur. *VMLinPgNum* is the address in the 1Meg V86 address space of the VM where the map will start (this is a page number, thus linear address A0000h = page A0h). Alignment considerations of this address (beyond 4K alignment) are the responsibility of the caller. Map addresses below page 10h, or above 10Fh will cause an error. *nPages* is the number of pages to map. The physical region is assumed to be contiguous (thus if mapping three pages, they will be PhysPage, PhysPage+1 and PhysPage+2 in that order). If the physical region is not contiguous, you will have to issue multiple calls in succession. There are currently no bits defined in the flags, this parameter must be set to 0.

Return Value Returns a nonzero value if the map is succesful, returns 0 value if the map was unsuccesful (invalid *VMHandle*, map range illegal).

Comments You are warned to be careful with this call. Very strange things will happen if you specify a physical region which is unoccupied, or belongs to some other device.

The page attributes for these pages will be P_USER+P_PRES+P_WRITE. P_DIRTY and P_ACC will be cleared by the call. PG_TYPE will be set to PG_SYS.

The intent of **PhysIntoV86** support for pages between page 10h and FirstV86Page is to support WIN386 devices which have **Allocate_Global_V86_Data_Area** a **GVDAPageAlign** region. Use of mapping in this region to other addresses can easily crash the system and should be avoided.

Regions which span across **FirstV86Page** are not allowed.

The reason for the page 10h limitation is that on most versions of the Intel 80386 CPU there is an errata which prevents you from setting up a Linear != Physical address mapping in the first 64k of the address space.

TestGlobalV86Mem

```
unsigned TestGlobalV86Mem(VMLinAddr,nBytes,flags)
   unsigned VMLinAddr;
   unsigned nBytes;
   unsigned flags;
```

Some WIN386 devices wish to test whether a given piece of V86 address space is LOCAL to a particular VM, or GLOBAL. The reason for this test is that GLOBAL V86 address ranges are valid and identical in ALL VM contexts, while LOCAL V86 address ranges are valid in only one VM context. This difference can yield optimizations. For instance, operations involving GLOBAL address ranges will typically not need to be "virtualized" in any way since the range is valid and addressable in ALL VM contexts. LOCAL address range operations may have to be "virtualized" though since it is possible for a piece of Virtual Mode code to try and use the address in the "wrong" VM context where the address range is invalid, or points to the wrong memory. This call can be used to test whether a V86 address range is GLOBAL or LOCAL. *VMLinAddr* is the linear address of the first byte of the V86 address range. This address is relative to the standard WIN386 RING 0 DS (ie. the linear address of 02C1:0FC5 would be 02C10 + 0FC5 = 3BD5). *nBytes* is the length of the V86 address range in bytes. There are currently no bits defined in the flags, this parameter must be set to 0.

Return Value

Returns 0 if the address is not a valid V86 address range, or the address range is LOCAL. Returns 1 if the address range is GLOBAL. Returns 2 if the address range is partly LOCAL and partly GLOBAL (range overlaps a GLOBAL/LOCAL boundary). Returns 3 if the address range is GLOBAL but overlaps with an Instance data region.

Comments

The distinction between GLOBAL and INSTANCE is rather subtle because INSTANCE pages are "physically global" even though their content is LOCAL. The physical address of instance data pages never changes, thus instance pages are GLOBAL in the physical address sense. The content of instance data regions is per VM though which means they are LOCAL in the sense of "what is in them".

The MMGR does not know any of the specifics about what is going on in the regions above FirstV86Page. This routine will return LOCAL for all regions above FirstV86Page, INCLUDING the A0-FF adapter/ROM BIOS area. Some pieces of this region may actually be GLOBAL in terms of how they are used, but this service doesn't know any of the details so it cannot determine this.

19.6 Looking At Physical Device Memory in Protected Mode

VxDs, such as virtual display drivers, that have a certain region of physical address space associated with them, such as Video Memory, need a way to look at the device-specific memory when the device is running. The method by which this is done is by using a service that returns the correct linear address (relative to the standard Ring 0 **DS**).

MapPhysToLinear

```
unsigned MapPhysToLinear(PhysAddr,nBytes,flags)
   unsigned PhysAddr;
   unsigned nBytes;
   unsigned flags;
```

PhysAddr is the physical address of the start of the region to be looked at. This is simply the 32 bit physical address, there are no alignment considerations. Physical addresses start at 0, thus the address of physical page 0A0h is 0A0000h. *nBytes* is the length of the physical region in bytes starting from *PhysAddr*. This parameter is used to verify that the entire range is addressable. There are currently no bits defined in the *flags*, this parameter must be set to 0.

Return Value

Returns the RING 0 DS offset of the first byte of the physical region. Will return 0FFFFFFFFh if the specified range is not addressable.

WARNING You are warned to be careful with this method. Use of this for purposes beyond looking at device specific physical memory is extremely dangerous and is not approved.

Comments Physical addresses do not move. It is perfectly fine to get the linear address of a physical region at Device_Init device call time and then use it later. You do not have to keep recalling **MapPhysToLinear** every time you want to look at the region.

For instance to look at physical page A0h you would do this:

```
VMMCall _MapPhysToLinear,<0A0000h,10000h,0>
```

DS:[EAX] now addresses this physical page. Physical memory is mapped contiguously at this selector so Page 0A1h would be 4096 bytes beyond the above address.

19.7 Data Access Services

These services are used to get the contents of public memory manager variables. Access to these variables is done via calls to support the DynaLink architecture of WIN386. All of these services return the value of the associated variable in EAX.

GetFirstV86Page

```
unsigned GetFirstV86Page()
```

This call returns the page number of the first page of VM specific V86 memory.

Comments **FirstV86Page** MOVES during device initialization. Do not get the value at device init time, and then use it later, as the value is invalid.

GetNulPageHandle

```
unsigned GetNulPageHandle()
```

This call returns the memory handle of the system NUL page.

GetAppFlatDSAlias()

```
unsigned GetAppFlatDSAlias()
```

This call returns a selector which can be used by protected mode applications to look at the same data that the standard WIN386 RING 0 DS looks at. This is useful when a WIN386 devide driver wishes to provide a protected mode service to applications and wants the application to be able to address the same memory that the WIN386 device driver does.

Comments

This selector is *read only*. This is so that the WIN386 address space is protected from a misbehaved application. IT is not recommended that you build a read/write version of this selector. If the application needs to WRITE you should build a descriptor with a much more restricted Base and Limit so that the application can only modify those things which it is allowed to modify.

This selector is RPL = DPL = Protected Mode Application Privilege. NOTE that a WIN386 device driver can also use this selector if desired even though the devices run at a different privilege level. Its type is "USE 16", this doesn't mean much since it is a data selector.

This is a GDT selector.

WARNING You must not do a Free_GDT_Selector on this selector. It is not protected, and so it will get freed. Then anyone using it will fault and crash the system. This selector is provided to prevent multiple devices from creating multiple versions of the same selector and wasting GDT entries unnecessarily.

Notice that enhanced Windows is "USE 32", therefore a protected application, which is "USE 16", will have to use the DB 67h addressing mode override on its instructions to get 32 bit addressing (MASM will do this for you automatically if you set things up correctly).

This service can be used to discover what protection ring Protected Mode applications run at by doing a LAR on the returned selector. Be very careful about what you do with this bit of information.

19.8 Special Services For Protected Mode APIs

These services are provided to support VxDs that need to manipulate protected-mode address space. For example, applications running in protected mode need a way to map regions of protected-mode, segmented address space into the virtual machine's virtual 8086 context. A specific example is the MS-DOS INT 21 API. The data pointed to on the INT 21 calls needs to be mapped into the VM's V86 address space so that MS-DOS can access it and perform the requested operation.

WARNING Do not use these services for purposes other than their intended use. These calls can be quite dangerous and can result in strange behavior or crashes if misused.

LinMapIntoV86

```
unsigned long LinMapIntoV86(HLinPgNum,VMHandle,VMLinPgNum,nPages,flags)
   unsigned HLinPgNum;
   unsigned VMHandle;
   unsigned VMLinPgNum;
```

```
unsigned nPages;
unsigned flags;
```

NOTE Please be advised that the following description has been identified as out of date in some respects though updated information was unavailable at time of printing.

This call is provided to assist the interface address mapper functions. Its purpose is to provide a way for the address mapper to map regions of protected mode address space into a VM V86 address space so that API calls can be performed. This calls operation is very similar to **MapIntoV86**, the difference being that instead of taking a memory handle, it takes a linear address. The call duplicates the memory map down into the indicated VM's V86 address range. *HLinPgNum*, together with *nPages*, indicates the region of protected mode address space, or V86 address space that is to be mapped. This is a page number, linear address 60610000h would be passed in as 60610h. As with **MapIntoV86** there are implied **PageLock** and **PageUnlocks**. Note that the linear address is relative to the standard WIN386 Ring 0 DS selector. The *VMHandle* parameter must be a valid *VM handle* and indicates the V86 space into which the map is to occur. *VMLinPgNum* is the address in the 1Meg VM V86 address space where the map will start (this is a page number, thus linear address 60000h = page 60h). Alignment considerations of this address (beyond 4K alignment) are the responsibility of the caller. Map addresses below page 10h, or above 10Fh will cause an error. *nPages* is the number of pages to map. Note that if *HLinPgNum* is a V86 page number (at the LOW V86 address (at the LOW V86 address <= page 100h) the call does nothing except return the *HLinPgNum* parameter in EDX. There are currently no bits defined in the flags. This parameter must be 0.

Return Value

The return value is a 64 bit long which is actually two 32 bit DWORDS. The Low DWORD (EAX) is a nonzero value if the map is succesful, returns 0 in eax if the map was unsuccesful (invalid address range, invalid *VMHandle*, map range illegal, size discrepancy, insufficient memory for implied **PageLock**). The High DWORD (**EDX**) is only valid if **EAX** is nonzero. It is set to the *VMLinPgNum* parameter if the *HLinPgNum* parameter was not a LOW V86 space address, otherwise it is set to the *HLinPgNum* parameter. In short, **EDX** is the V86 address where the memory is mapped.

Comments

As with **MapIntoV86** there is an implied **PageLock** which is performed on all of the pages mapped. This is consistent with the fact that V86 memory cannot be Demand Paged while the VM is in a runable state. Whenever the V86 memory mapping is changed via **LinMapIntoV86**, the previous memory that was mapped in the VM is unlocked. The correct way to think of this is that there is an implied **PageLock** whenever memory is mapped into a V86 context, and an implied **PageUnlock** whenever it is "unmapped" from the V86 context. This "unmapping" can occur when: A different handle (including the NulPageHandle) is MapIntoV86ed to the region, or a **PhysIntoV86** is performed to the region.

The V86 region mappped into by this call should be **MapIntoV86ed** with the NulPageHandle when the V86 mapping region is no longer needed. There is nothing to prevent you from mapping the same protected mode linear address into multiple places in a VM, or into multiple VMs. Such operations are not particularly advisable though. For one thing, the reporting of memory owned by a VM will be disturbed.

The reason this call exists is because a protected mode API mapper does not have access to the memory handles associated with the various regions of protected mode address space. VxDs which do have access to the memory handles of the memory to be mapped should be using **MapIntoV86** to map the memory, not this routine.

For regions in the Physical addressing region this call will convert into a **PhysIntoV86** call.

For regions in the HIGH VM Linear addressing region this call will perform a map of the memory from one VM into another VM (or into a different location in the same VM). NOTE CAREFULLY: The intent of this support is to provide a way for the V86MMGR device to map a region of V86 address space which is currently LOCAL to one VM into a GLOBAL region that is addressable by all VMs. This type of API is needed by network API mappers. Do not use this capability in your VxD, use the V86MMGR service. The details of this aspect of operation will change in a later release and code using the old method will not function properly.

The page attributes for these pages will be P_USER+P_PRES+P_WRITE. P_DIRTY and P_ACC will be cleared by the call. PG_TYPE will be set to whatever the type of the pages are at its protected mode linear address.

The intent of **LinMapIntoV86** support for pages between page 10h and **FirstV86Page** is to support WIN386 devices which have **Allocate_Global_V86_Data_Area** a GVDAPageAlign region. Use of mapping in this region to other addresses can easily crash the system and should be avoided.

Regions which span across **FirstV86Page** are not allowed.

The reason for the page 10h limitation is that on most versions of the Intel 80386 CPU there is an errata which prevents you from setting up a Linear != Physical address mapping in the first 64k of the address space.

LinPageLock

```
unsigned LinPageLock(HLinPgNum,nPages,flags)
   unsigned HLinPgNum;
   unsigned nPages;
   unsigned flags;
```

This call is provided to assist the interface address mapper functions. Its purpose is to provide a way for the address mapper to lock regions of protected mode address space so that API calls can be performed. This calls operation is very similar to **PageLock**, the difference being that instead of taking a memory handle, it takes a linear address. *HLinPgNum*, together with *nPages*, indicates the region of protected mode address space that is to be locked. This is a page number, linear address 60610000h would be passed in as 60610h. Note that the linear address is relative to the standard WIN386 Ring 0 DS selector.

Current flags bits:

```
PageLockedIfDP          EQU     00000000000000000000000100000000B
```

All unused bits must be zero. **PageLockedIfDP**, if set, indicates that the lock only needs to be done if the **PageSwap** device is not direct to hardware. In the case where the **PageSwap** device is of type two (direct to hardware), calls to this routine with **PageLockedIfDP** set are effectively NOPs. See the **PageAllocate** documentation for a description of the different **PageSwap** device types and their relevance.

Return Value Returns a nonzero value if the lock is succesful, returns 0 value if the lock was unsuccesful (invalid address range, insufficient memory for lock).

Comments SEE **PageLock**.

LinPageUnLock

```
unsigned LinPageUnLock(HLinPgNum,nPages,flags)
   unsigned HLinPgNum;
   unsigned nPages;
   unsigned flags;
```

This call is provided to assist the interface address mapper functions. Its purpose is to provide a way for the address mapper to unlock regions of protected mode address space after API calls are performed. This calls operation is very similar to **PageUnLock**, the difference being that instead of taking a memory handle, it takes a linear address. *HLinPgNum*, together with *nPages*, indicates the region of protected mode address space that is to be unlocked. This is a page number, linear address 60610000h would be passed in as 60610h. Note that the linear address is relative to the standard WIN386 Ring 0 DS selector.

Current flags bits:

```
PageLockedIfDP          EQU     00000000000000000000000100000000B
PageMarkPageOut         EQU     00000000000000000010000000000000B
```

All unused bits *must be zero*. **PageLockedIfDP**, if set, indicates that the unlock only needs to be done if the **PageSwap** device is not direct to hardware. In the case where the **PageSwap** device is of type two (direct to hardware), calls to this routine with **PageLockedIfDP** set are effectively NOPs. See the **PageAllocate** documentation for a description of the different **PageSwap** device types and their relevance. **PageMarkPageOut**, if set, indicates that if this unlock actually does unlock the pages (lock count goes to 0) the pages are to be made prime candidates for page out. This flag should only be set if it is unlikely that these pages are going to be touched for a while. Effectively what this does is clear the P_ACC bits of the pages which causes them to be first level page out candidates.

Return Value Returns a nonzero value if the unlock is succesful, returns 0 value if the unlock was unsuccesful (invalid address range).

Comments SEE **PageUnLock**.

PageCheckLinRange

```
unsigned PageCheckLinRange(HLinPgNum,nPages,flags)
   unsigned HLinPgNum;
   unsigned nPages;
   unsigned flags;
```

This call is provided to assist the interface address mapper functions. Its purpose is to provide a way for the address mapper to validate an intended range for **LinPageLock** or **LinMapIntoV86**. Sometimes a MAXIMUM length range is specified because the true range is unknown. This call will return an adjusted *nPages* argument which will be adjusted down in size if the specified range crosses an unreasonable boundary. *HLinPgNum*, together with *nPages*, indicates the region of protected mode address space that is to be checked. This is a page number, linear address 60610000h would be passed in as 60610h. Note that the linear address is relative to the standard WIN386 Ring 0 DS selector. There are currently no bits defined in the flags, this parameter must be 0.

Return Value Returns an adjusted *nPages* agrument. This will be *zero* if the range is totally unreasonable, and will return *nPages* if no adjustment was needed.

Comments The end of a handle is a boundary that will result in an adjustment.

SelectorMapFlat

```
unsigned SelectorMapFlat(VMHandle,Selector,flags)
   unsigned VMHandle;
   unsigned Selector;
   unsigned flags;
```

This call is provided to assist the interface address mapper functions. Its purpose is to provide a way for the address mapper to get the RING 0 DS offset of the base of a particular GDT or LDT selector. This call assists the address mapper in converting a Selector:Offset16 or Selector:Offset32 pointer into its "flat model" linear address which can then be passed to **LinMapIntoVM**. Selector is a GDT or LDT selector (note that the argument is a DWORD not a WORD) value to get the base address of. The *VMHandle* parameter is ignored if Selector is a GDT selector. If Selector is an LDT selector, then *VMHandle* indicates the appropriate VM context for the Selector. There are currently no bits defined in the flags, this parameter must be 0.

Return Value Returns the linear address of the base of the selector if succesful, returns FFFFFFFFh if it is unsuccesful (invalid selector).

Comments

You can pass this routine the standard WIN386 RING 0 DS selector, and it will return 0 as the base. This is a silly thing to do, but it does work.

The *VMHandle* parameter must be valid for LDT selectors.

SetResetV86Pageable

```
unsigned SetResetV86Pageable(VMHandle,VMLinPgNum,nPages,flags)
unsigned VMHandle;
unsigned VMLinPgNum;
unsigned nPages;
unsigned flags;
```

This call allows the normal locking/unlocking behavior associated with a specific range of V86 memory to be modified. *VMHandle* is the VM in which the behavior is being modified. *VMLinPgNum* is the address in the 1Meg V86 address space where the behavior modification will start (this is a page number, thus linear address 60000h = page 60h). Alignment considerations of this address (beyond 4K alignment) are the responsibility of the caller. Map addresses below **FirstV86Page**, or above 100h will cause an error. *nPages* is the number of pages to modify the behavior of. Normally a **MapIntoV86** causes the memory that is mapped to be locked. In the case where this particular VM is currently running a Protected Mode application, it is desirable to undo the lock, and change this normal lock/unlock behavior. This allows those unused pieces of the V86 address space to be paged out and the memory they are using to be used by someone else. Note that we can only undo this normal behavior because the behavior of the protected mode application is well known. In particular, we know that none of the V86 memory that is being unlocked contains code that is executed, or data that is touched, at interrupt time (including software interrupt time). The typical use of this call is by the WIN386 device which loads a protected mode application. When the PM app is loaded, the device calls **SetRestV86Pageable** with the **PageSetV86Pageable** bit set on those pieces of the V86 address space above **FirstV86Page** which can be unlocked; this is typically all of the V86 memory above FirstV86Page which is currently DOS Free. NOTE that DOS data areas such as the 100h byte Program Header Prefix *must not be included* in the ranges because they are accessed by DOS. Similarly, when the Protected Mode application Exits, the application loader calls **SetResetV86Pageable** with PageClearV86Pageable set, on the V86 memory it had initially modified during the load.

The other aspect of the behavior that can be modified has to do with the “other memory” (the memory that is *not* V86Pageable) in the VM. Normally this memory is locked, except when the pager is type 2 (direct to hardware). Not locking the V86 memory allows VM’s V86 pages to also be Demand Paged. This has the benefit of allowing DOS applications to also run in a Demand Paged environment. Sometimes though, this is an undesired behavior because of the paging latency which it introduces in the VM. The V86IntsLocked bit of a VM allows this aspect to be controled. Setting the V86IntsLocked behavior causes the “other memory” to *always* be locked, even if the pager is type 2. Setting this behavior has two important effects:

- There is never any "paging latency" while the virtual mode code in this VM is running. This prevents time critical V86 code from having its timing severly disturbed due to the paging overhead.
- The paging device can enable interrupts in this VM when it is performing paging operations because *it knows* that a nested page fault will not occur from this VM since all of its interrupt time code is always locked.

Current flags bits:

```
PageSetV86Pageable          EQU     00000000000000000000001000000000B
PageClearV86Pageable        EQU     00000000000000000000010000000000B
PageSetV86IntsLocked        EQU     00000000000000000000100000000000B
PageClearV86IntsLocked      EQU     00000000000000000001000000000000B
```

All unused bits *must be zero*. **PageSetV86Pageable**, if set, indicates that the normal locking behavior of **MapIntoV86** is to be disabled (V86 memory can be paged) for the indicated region. **PageClearV86Pageable**, if set, indicates that the normal locking behavior is to be enabled on the indicated region. **PageSetV86IntsLocked**, if set, indicates that the "lock all V86 memory that is not V86Pageable regardless of pager type" behavior is to be enabled. PageClearV86IntsLocked, if set, indicates that the "lock all V86 memory that is not V86Pageable regardless of pager type" behavior is to be disabled. *Note* that only *one* of these bits can be set on a call. Setting more than one bit will result in an error. There are two bits in **CB_VM_Status** that indicate the current state of these behaviors:

```
VMStat_PageableV86          EQU     00000000000000000000100000000000B
VMStat_V86IntsLocked        EQU     00000000000000000001000000000000B
```

The **VMStat_PageableV86** bit is set if any regions behavior has been modified (there is at least one non zero bit in the array returned by **GetV86PageableArray**). The **VMStat_V86IntsLocked** bit is set if the "lock regardless of pager type" behavior has been enabled in this VM.

Return Value

Returns non-zero value if the set or clear worked, zero if the current state of the VM was not consistent with the call (invalid VMHandle, **VMStat_PageableV86** or **VMStat_V86IntsLocked** state inconsistent with setting of **PageSet/ClearV86Pageable** or **PageSet/ClearV86IntsLocked** bit in flags, range invalid) or the lock of the memory associated with **PageClearV86Pageable** or **PageSetV86IntsLocked** failed.

Comments

The intent of this call is to better support Protected mode applications running in a VM, not to allow you to randomly make v86 parts of VMs pageable! Do not issue this call on a VM unless you are loading a Protected mode app into it.

The V86MMGR device makes a **PageSetV86IntsLocked** call on VMs which are created with their base memory specified as *locked*.

Extreme care must be used when manipulating the PageableV86 behavior of regions above A000:0. This should not be done unless the region is GLOBAL or LOCAL **Assign_Device_V86_Pages** owned by the caller.

There is no REGION associated with **PageSetV86IntsLocked** and **PageClearV86IntsLocked** calls. The IMPLIED region is always "everything that isn't V86Pageable". For this reason the *HLinPgNum* and *nPages* arguments should be set to 0 on these calls.

VMM.INC contains equates for all of the flag bits described, use the equates.

GetV86PageableArray

```
unsigned GetV86PageableArray(VMHandle,ArrayBufPTR,flags)
unsigned VMHandle;
unsigned ArrayBufPTR;
unsigned flags;
```

This call is used to obtain a copy of the bit array of pages whose behavior has been modified via **SetResetV86Pageable.** This allows the caller to determine which regions of the VM V86 address space have had the normal lock/unlock behavior modified. VMHandle specifies the VM to get the bit map of. ArrayBufPTR points to a buffer large enough to contain the array. The assignment array is an array of 100h bits, one bit for each page in the range 0-100h. Thus the size of the array is ((100h/8)+3)/4 = 8 DWORDS. Bits in the array which are set (=1) indicate pages whose normal lock/unlock behavior is disabled, bits which are clear (=0) indicate pages whose behavior is normal. Thus to test the bit for page number N (0 <= N <= 0FFh) you could use code like this:

```
mov     ebx, N MOD 32                         ; Bit number in DWORD
mov     eax, N / 32                           ; DWORD index into array
bt      dword ptr ArrayBufPTR[eax*4],ebx; Test bit for page N
jnc     short PageNormal
    PageModified:
```

Note that this code is mearly intended to illustrate how the bit array works. This code is not the most efficient, or the only way to implement this test. There are currently no bits defined in the flags, this parameter must be set to 0.

Return Value Returns non-zero if succesfull, returns zero if the bit array could not be returned (Invalid VMHandle).

Comments Making this call on a VM whose **VMStat_PageableV86** bit is clear is not an error, it simply returns a bit array whose bits are all 0.

PageDiscardPages

```
unsigned PageDiscardPages(LinPgNum,VMHandle,nPages,flags)
unsigned LinPgNum;
unsigned VMHandle;
unsigned nPages;
unsigned flags;
```

This call is provided to assist management of PM applications by providing a way to mark pages as "no longer in use". What this does is allow regions which were previously "in use" to be "discarded". This means that the page does not have to be "paged in" to make it present, thus eliminating the disk access required for the page in. LinPgNum and nPages together specify the range to be discarded. LinPgNum is a page NUMBER. If LinPgNum is < 110h, or at a VM high linear address, then the range lies in a VM and the *VMHandle* parameter specifies the VM. In this case, all pages of the range must be marked V86Pageable or the call will fail. Pages in the range which are not present or are locked are ignored, this call effects only demand pageable pages.

Current flags bits:

```
PageZeroInit        EQU     00000000000000000000000000000001B
PageDiscard         EQU     00000000000000010000000000000000B
```

Setting PageDiscard indicates that a full discard is to take place, the P_ACC and P_DIRTY bits in the page table entrys for the pages are both cleared. If PageDiscard is clear, all the call does is clear the P_ACC bit in the page table entrys for the pages making them primary page out candidates (the DIRTYness and content of the pages is preserved in this case). Setting PageZeroInit is relevant only if PageDiscard is also set, and it indicates that the pages are to be marked "zero the contents of this page the next time it is paged in". In this case this subsequent page in is a NOP since the pages have been discarded, this simply causes the pages to come back in with a known value (0) in them instead of random garbage.

Return Value Returns a non-zero value if successful, otherwise it returns zero (invalid range or VM handle).

Comments The Flag bit equates are defined by including VMM.INC, please use the equates.

19.9 Instance Data Management

The purpose of these services is to provide a means of identifying to the system those areas of virtual 8086 mode memory (V86 memory) that contain per Virtual Machine or "Instance" data. Each of the VMs in the system has its own, private instance of this data and anything the VM does to the values in these locations has no effect on other VMs since the values are different in each VM.

NOTE All of these calls use the USE32 C calling convention. The true name of the procedure has an underscore in front (i.e., **AddInstanceItem** is actually **_AddInstanceItem**), and the arguments are pushed right to left (unlike the PL/M calling convention used by Windows, which is left to right). The return value(s) is returned in C standard **EDX:EAX**. It is the responsibility of the *caller* to clear the arguments off the stack. Registers **EAX**, **ECX**, and **EDX** are changed by calls. Registers **DS**, **ES**, **EBP**, **EDI**, **ESI**, and **EBX** are preserved.

AddInstanceItem

```
unsigned AddInstanceItem(InstStrucPTR,flags)
   unsigned InstStrucPTR;
   unsigned flags;
```

This call is used to identify a region of instance data in the V86 address space. *InstStrucPTR* is a pointer to an instance data identification structure which has this form:

```
InstDataStruc struc
   InstLinkF        dd    ?      ; RESERVED SET TO 0
   InstLinkB        dd    ?      ; RESERVED SET TO 0
   InstLinAddr      dd    ?      ; Linear address of start of block
   InstSize         dd    ?      ; Size of block in bytes
   InstType         dd    ?      ; Type of the block
InstDataStruc ends
```

The InstLinkF and InstLinkB fields are filled in by the Instance data manager and cannot be used by the caller. InstLinAddr defines the start of the block of instance data, NOTE THAT THIS IS NOT IN SEG:OFFSET FORM, it is a linear address. Thus the correct value for 40:2F would be 42F. InstSize is the size of the instance data block in bytes starting at InstLinAddr. InstType defines one of two types of instance data:

```
INDOS_Field     equ    100h   ; Bit indicating INDOS switch requirements
ALWAYS_Field    equ    200h   ; Bit indicating ALWAYS switch requirements
```

ALWAYS_Field type indicates that the field must always be switched when a VM is switched. All instance data sepcified by VxDs should be of this type. INDOS_Field type is reserved for special types of DOS internal data which only need to be switched with the VM if the VM is currently INDOS.

There are currently no bits defined in the flags, this parameter must be set to 0.

Return Value Returns nonzero value if the instance data block was succesfully added to the instance list, zero if the block was unsuccesful added (This is probably a FATAL error).

NOTE There are two basic ways to allocate the space for the InstDataStrucs pointed to with InstStrucPTR. The first is to simply staticly allocate them in the INIT data segment. The space they occupy will then be reclaimed when the INIT space is reclaimed. The other way is to allocate them on the System heap using HeapAllocate. The space can then be freed by HeapFreeing all of the heap handles in the device Sys_VM_Init code which is called after all of the system initialization (including the instance data initialization) is done.

WARNING If you allocate space for InstDataStrucs on the heap you must be sure NOT to HeapReAllocate the heap blocks after passing the address to AddInstanceItem because this will invalidate the InStrucPTR value you previously passed to AddInstanceItem.

NOTE This routine is in the init segment of WIN386. It can therefore only be called during system initialization. Trying to call it after system initialization and the system INIT segment space has been reclaimed will result in a fatal page fault.

Once this call is made, the caller **must not** ever touch the InstDataStruc pointed to again. The caller has passed control of this data block to the instrance data manager and tampering with it will result in the instance data manager failing to identify the instance data correctly.

Note that only one, contiguous region of instance data can be identified with each structure. It is a good idea for the caller to coalesce adjacent blocks of instance data it is identifying in order to cut down the call overhead and data space requirements, but this is not required.

There is a declaration of the InstDataStruc data structure in VMM.INC.

MMGR_Toggle_HMA

```
unsigned MMGR_Toggle_HMA(VMHandle,flags)
        unsigned VMHandle;
        unsigned flags;
```

This call is an interface to the Instance data manager which allows devices such as the V86MMGR XMS device to control the behavior of the "highmem" memory area, or "HMA", of a VM (V86 linear pages 100h through 10Fh). Any device which wishes to modify the "1Meg Address Wrap" behavior of a VM MUST use this call to inform the Instance data manager what is going on. This is because the Instance manager must know whether 1Meg Address Wrap is on or off to manage the instance data correctly for a VM. *VMHandle* is a valid WIN386 VM handle which indicates the VM to which the call is to be applied. Current flags bits:

```
MMGRHMAPhysical        EQU     00000000000000000000000000000001B
MMGRHMAEnable          EQU     00000000000000000000000000000010B
MMGRHMADisable         EQU     00000000000000000000000000000100B
MMGRHMAQuerry          EQU     00000000000000000000000000001000B
```

All unused bits must be zero. ONe, and only one of MMGRHMAEnable, MMGRHMADisable, MMGRHMAQuerry BITS must be specified, the call will have random results if this is not true. MMGRHMAPhysical bit is a modifier which modifies the operation of the MMGRHMAEnable bit: See discussion of MMGRHMAEnable. MMGRHMADisable, if set, causes the Instance manager to restore the normal Wrap mapping for pages 100 through 10F thus Disabling the HMA. This is a REMAP of pages 00h through 0Fh of the VM and causes the VMs address space to "wrap" back to address zero for addresses >1Meg as it does on an 8086 processor. MMGRHMAEnable, if set, disables 1Meg address wrap in the VM, thus Enabling the HMA. Exactly what this does is controlled by the MMGRHMAPhysical bit. If MMGRHMAPhysical is set, MMGRHMAEnable causes PHYSICAL pages 100h through 10Fh to be mapped in Linear pages 100h through 10Fh of the VM consistent with the operation of a Global HMA which is shared by all VMs. If MMGRHMAPhysical is not set, Linear pages 100h through 10Fh will be marked as not present System Pages in the VM. It is then up to the CALLER to map some other memory handle into this region of the VM after this call. This is consistent with the operation of a per VM HMA. Note that if the VM accesses these pages before this mapping is set up, an erroneaous page fault will occur which will crash the VM, or the system. MMGRHMAQuerry, if set, returns the current state of the HMA in the VM.

Return Value

This call has no return value unless MMGRHMAQuerry was specified in the flags. In this case the call will return value 0 if the HMA is Disabled (1Meg address wrap is enabled), and it will return a nonzero value if the HMA is Enabled (1Meg address wrap is disabled).

Comments

This call is reserved for the V86MMGR XMS device. Other devices should not be using this call. Modifying the Wrap state of a VM without the V86MMGR XMS device knowing about it will probably result in a state error and a crash.

The device issuing this call must be a device which has succesfully Globally or Locally **Assign_Device_VM_Paged** pages 100h through 10Fh in the indicated VM. This is not a call which multiple devices should make for a VM as doing so will cause confusion between the devices.

When VMs are created, they are created with the HMA Disabled (1Meg Address Wrap enabled) consistent with normal operation on an 8086 processor. The device responsible for the HMA in a VM must adjust this in its **Create_VM** device call if needed.

Note that no distinction is drawn on the MMGRHMAQuerry return between MMGRHMAPhysical being specified, or not specified on a previous MMGRHMADisable call.

NOTE Instance data is not allowed in the hma.

The flag bit equates are in VMM.INC, please use the equates.

19.10 Looking At V86 Address Space

From time to time, VxDs may wish to look at or modify some piece of the virtual 8086 mode address space of a VM that is not the current VM. The documented way to do this is as follows.

CB_High_Linear

There is a Control Block variable which is a linear address of the start of the VM's address space. Thus to look at VM linear adress 40:17 with EBX being the VM Handle of the VM you're interested in you would do this:

```
mov     esi,(40h SHL 4) + 17h
add     esi,[ebx.CB_High_Linear]
```

ESI now points to this location in the V86 address space. This can be used to look at, modify any V86 address including instance data addresses.

NOTE No code should EVER touch a part of V86 address space at its "low" address (>=0,<=400000h) EVEN FOR THE CURRENT VM. There is NO REASON to do this, use CB_High_Linear in ALL cases to look at V86 addresses.

Chapter 20 I/O Services and Macros

This chapter documents the services available for I/O. Also included are two macros and a discussion explaining their usefulness.

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

When a virtual machine executes an instruction that reads or writes data from an I/O port, the 80386 looks up the port number in the I/O Permission Map (IOPM). If the corresponding bit in the IOPM is set, then the instruction will cause a protection fault.

Enhanced Windows provides services that virtual devices use to trap I/O. The first thing a virtual device must do is hook the port while the device is being initialized. This is done by calling a service called **Hook_IO_Port**. It takes two parameters: the number of the I/O port to hook and the address of a callback procedure.

When **Hook_IO_Port** is called, enhanced Windows sets the appropriate bit in the I/O permission map and registers the callback procedure. Whenever a VM accesses the port, the VMM will call the procedure with the following parameters:

EBX = Handle of VM that accessed the port
EDX = Port number
ECX = Type of I/O
If VM is outputing data to the port then
EAX/AX/AL = Output value

20.1 Handling Different I/O Types

The value passed in ECX determines the type of input or output as specified by Table 20.1.

Table 20.1 I/O Register Values

Value	Type of input/output
00H	Byte input
04H	Byte output
08H	WORD input
0CH	WORD output
10H	DWORD input

14H	DWORD output

Masks that apply only to string I/O are shown in Table 20.2.

Table 20.2 String I/O Register Values

Value	Type of input/output
20H	String I/O
40H	Repeated string I/O
80H	32-bit addressing mode string I/O
100H	Reverse string I/O (VM's direction flag is set)

For all string I/O operations, the high WORD of **ECX** contains the segment for the string I/O. This allows VxDs to ignore the issues of segment overrides on these instructions; VMM has already determined the correct segment value. Thus, a value of 3247016CH would specify that the VM is doing word reverse repeated string output to 3247:DI.

For example:

High word = segment 3247
0Ch = Word output
20h = String I/O
40h = Repeated string I/O
100h = Reverse I/O

It would be unreasonable to expect every VxD to support 48 different types of I/O. Therefore, the VxD environment only requires VxDs to support byte input and output, even though a VxD can directly support any type of I/O that is appropriate. For example, there is no reason for the Virtual Printer Device (VPD) to support WORD input and output since printer ports are only 8-bits wide.

However, there are 16-bit VxDs for 16-bit ports that must directly support WORD I/O as well as byte I/O.

Furthermore, devices such as disk drives might need to directly emulate string I/O for some ports to achieve acceptable performance. A device can emulate some types of I/O and ignore others.

But what happens if someone does WORD string output to a printer port? You canot just throw the I/O away! For this reason, enhanced Windows has a catch-all routine called **Simulate_IO** that converts I/O into something the virtual device can understand. Notice in the port trap code of the VPD example that entry points start with the **Emulate_Non_Byte_IO** macro. This macro generates the following code:

```
cmp     ecx, 4
jbe     SHORT Foo
```

```
        VMMjmp  Simulate_IO

Foo:
```

So, if a VM attempted to do non-repeated forward **word** string I/O, the following sequence of calls to the VPD trap code would be issued:

Call VPD trap with:

EBX = VM handle
EDX = 358h (Port #)
ECX = 23A8002Ch (String I/O from segment 23A8h)
EBP = Client register structure

VPD jumps to **Simulate_I/O** which calls VPD again with:

EBX = VM handle
EDX = 358h (Port #)
ECX = 0Ch (0Ch = Word output)
AX = Word output
EBP = Client register structure

VPD jumps to **Simulate_I/O** which calls VPD again with:

EBX = VM handle
EDX = 358h (Port #)
ECX = 04h (04h = Byte output)
AL = Byte output
EBP = Client register structure

VPD then simulates the byte output and returns.

Notice that the high-order byte of the word output would be sent to the trap routine for VPD trap port # +1. So, if VPD is trapping port 358H, then word output to this port will be converted into byte output to ports 358H and 359H (exactly the way the hardware works).

20.2 I/O Macros

There are two useful macros for I/O trap routines. The first macro, **Emulate_Non_Byte_IO**, generates the following code:

```
        cmp     ecx, Byte_Output
        jbe     SHORT Is_Byte_IO
        VMMjmp  Simulate_IO
Is_Byte_IO:
```

Dispatch_Byte_IO, the second useful macro, takes two arguments. The first is the destination for byte input, and the second is the destination for byte output. This macro passes back all non-byte I/O to **Simulate_IO**. A typical I/O trap routine looks like the following example:

```
BeginProc VfooD_Trap_Data
 Dispatch_Byte_IO Fall_Through, VFood_Out_Data
  ...
 (Code for byte input)
 ...
 ret

VfooD_Out_Data:
 ...
 (Code for byte output)
 ...
 ret
EndProc VfooD_Trap_Data
```

Notice the special value **Fall_Through** that instructs the **Dispatch_Byte_IO** macro that byte input should fall through to the following code. You can substitute **Fall_Through** for either the input or output parameter (but not both) or specify two labels.

20.3 I/O Services

This section presents detailed information on each of the following I/O services in the following order:

- **Enable_Global_Trapping**
- **Disable_Global_Trapping**
- **Enable_Local_Trapping**
- **Disable_Local_Trapping**
- **Install_IO_Handler**
- **Install_Mult_IO_Handlers**
- **Simulate_IO**

Enable_Global_Trapping, Disable_Global_Trapping

Description These services enable and disable I/O port trapping in every VM. A callback hook must have been installed during initialization before either of these services is used.

The global trapping state is by default enabled. When a VM is created, it will be created with the current global trapping state.

Entry EDX = I/O port number

Exit None

Uses Flags

Enable_Local_Trapping, Disable_Local_Trapping

Description These services enable and disable I/O port trapping in a specific VM. A callback hook must have been installed during initialization before either of these services is used.

Entry **EBX** = VM handle
EDX = I/O port number

Exit None

Uses Flags

Install_IO_Handler (Initialization only)

Description This service installs a callback procedure for I/O port trapping and enables trapping for the specified port in all VM's. Only one procedure may be installed for each port.

When an I/O callback is installed, the default global trapping state is enabled. You can disable trapping of a port for every or specific VMs using the **Enable/Disable_Global_Trapping** and **Enable/Disable_Local_Trapping** services.

Entry **ESI** = Address of procedure to call
EDX = I/O port

Exit

```
If carry set then
    ERROR:  Port already hooked by another device or
    unable to hook any more ports (out of hooks)
else
    Port hooked successfully
```

Uses Flags

Callback **EBX** = Current VM handle
ECX = Type of I/O
EDX = Port number
EBP -> Client register structure

```
If output then
    EAX/AX/AL = Data output to port
else (input)
```

```
Callback procedure must return EAX/AX/AL for data input from
port
```

Install_Mult_IO_Handlers (Initialization only)

Description This service makes repeated calls to the **Install_IO_Handler** service with the entries in a table built using macros as follows:

```
Begin_Vxd_IO_Table  Table_Name
    Vxd_IO       <port#>, <procedure name>
    ...
    Vxd_IO       <port #>, <procedure name>
    Vxd_IO       <port #>, <procedure name>
 End_Vxd_IO_Table    Table_Name.
```

Entry **EDI** = Address of VxD_IO_Table

Exit

```
If carry set then
   ERROR: One or more ports already hooked by another device
           or unable to hook any more ports (out of hooks)
   EDX = Number of port that could not be hooked
 else
   Ports hooked successfully
```

Uses Flags

Callback

EBX = Current VM handle
ECX = Type of I/O
EDX = Port number
EBP -> Client register structure

```
If output then
    EAX/AX/AL = Data output to port
else (input)
    Callback procedure must return EAX/AX/AL for data input from
    port
```

Simulate_IO

Description This service is used to break complex I/O instructions into simpler types of I/O. An I/O handler should jump to this service using **VMMjmp Simulate_IO** whenever the handler is called with a type of I/O that it does not directly support. A typical I/O trap handler would start with code similar to the following:

```
Sample_IO_Handler:
        cmp     ecx, Byte_Output
        je      SHORT SIH_Simulate_Output
        jb      SHORT SIH_Simulate_Input
        VMMjmp  Simulate_IO
```

Since byte input is 0 and byte output is 4, a single compare can be used to determine if the I/O is byte input, output, or not supported. When **Simulate_IO** is invoked, it will break the I/O into simpler I/O types and recursively call **Sample_IO_Handler**.

For example, assume **Sample_IO_Handler** is the I/O trap handler for port 534H. If it was called with **ECX = Word_Output**, then it would immediately jump to the **Simulate_IO** service. **Simulate_IO** would then break the I/O instruction into byte output to ports 534H and 535H. When **Sample_IO_Handler** was called again, it would be able to virtualize the byte output to port 534H. The output to port 535H would be handled by another port trap routine, or, if there was not one installed, the output would be reflected directly to hardware port 535H.

Two macros, **Emulate_Non_Byte_IO** and **Dispatch_Byte_IO**, are provided as convenient ways to invoke this service.

Emulate_Non_Byte_IO is usually the first line of an I/O trap handler. It simply compares **ECX** to **Byte_Output** and, if it is greater, it jumps to the **Simulate_IO** service. For example:

```
Sample_IO_Handler:
        Emulate_Non_Byte_IO
        (Here ECX will be 0 for byte input or 4 for byte output)
```

Dispatch_Byte_IO is usually more convenient since it will also jump to the appropriate code for byte input or output. The macro takes two parameters. The first parameter specifies the label to jump to for byte input, and the second specifies the label to jump to for byte output. Either parameter (but not both) can have the special value **Fall_Through**, which specifies that the code to handle that I/O type immediately follows the macro. For example:

```
Sample_IO_Handler:
 Dispatch_Byte_IO Fall_Through, <SHORT  SIH_Output>
 (...Code here for handling byte input...)
 ret

SIH_Output:
        (...Code here for handling byte output...)
        ret
```

If, for efficiency reasons, you want to provide code to virtualize I/O other than byte input and output, test for the types that you can handle and then jump to this service to emulate other types of I/O.

Notice that the entry parameters to this service are identical to the parameters passed to your I/O trap routine. You should jump to this service using the **VMMjmp** macro with all of the registers in the same state as when your I/O trap routine was called (although you may modify **ESI** and **EDI** since they are not parameters).

Entry

EAX = Data for output instructions
EBX = Current VM handle
ECX = Type of I/O (same as passed to I/O trap routine)
EDX = I/O port
EBP -> Client Register Structure

Exit

All registers modified. If input, then **AX** or **EAX** will contain virtualized input value.

Uses

EAX, EBX, ECX, EDX, ESI, EDI, Flags

Chapter 21 VM Interrupt and Call Services

The VM Interrupt and Call Services supported by enhanced Windows are described in this chapter in the following order:

- **Build_Int_Stack_Frame**
- **Call_When_VM_Ints_Enabled**
- **Disable_VM_Ints**
- **Enable_VM_Ints**
- **Get_PM_Int_Type**
- **Get_V86_Int_Vector**
- **Get_PM_Int_Vector**
- **Hook_V86_Int_Chain**
- **Hook_PM_Int_Chain**
- **Set_PM_Int_Type**
- **Set_V86_Int_Vector**
- **Set_PM_Int_Vector**
- **Simulate_Far_Call**
- **Simulate_Far_Jmp**
- **Simulate_Far_Ret**
- **Simulate_Far_Ret_N**
- **Simulate_Int**
- **Simulate_Iret**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device (VxD) Programming Topics," for general discussions on VM Interrupts and Call Services.

Build_Int_Stack_Frame

Description This service will save the current **CS:IP** and flags on the VM's stack and, then, set the **CS:IP** to the value passed to the routine. The next time the VM is entered, the effect will be that an interrupt occurred, directing control to the procedure provided.

The procedure that is called must do an IRET to return.

Sample code:

```
VMMcall Begin_Nest_Exec
      mov      cx, [My_Private_VM_Proc_Segment]
      mov      edx, [My_Private_VM_Proc_Offset]
      VMMcall  Build_Int_Stack_Frame
      VMMcall  Resume_exec
VMMcall End_Nest_Exec
```

Entry **CX** = Code segment of procedure to call
EDX = Offset of procedure to call (high word must be 0 for 16-bit apps)

Exit None

Uses **Client_CS**, **Client_EIP**, **Client_Flags**, Flags

Call_When_VM_Ints_Enabled

Description If a VxD needs to be called when interrupts are enabled, it can use this service to be notified when the VM enables interrupts. If the current VM's interrupts are already enabled when this service is called, your callback procedure will be called immediately.

It is usually more convenient to use the **Call_Priority_VM_event** service instead of calling this service directly. However, this service is faster.

Entry **EDX** = Reference data
ESI = Offset of procedure to call

Exit None

Uses **Client_Flags**, Flags

Callback **EBX** = Handle of current VM
EDX = Reference data passed to this service
EBP -> Client register structure
Called procedure may destroy **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, and Flags

Disable_Vm_Ints

Description This service will disable interrupts during VM execution for the current virtual machine. This has the same effect as the VM executing a CLI instruction.

Entry None

Exit None

Uses Flags

Enable_VM_Ints

Description This service will enable interrupts during VM execution for the current virtual machine. This has the same effect as the VM executing an STI instruction.

If any VxDs have scheduled callback events using the **Call_When_Ints_Enabled** or **Call_Priority_VM_Event** services, then the callback procedure(s) will be called before this service returns.

Entry None

Exit None

Uses Flags

Get_PM_Int_Type

NOTE The description for this service has been identified as out of date and the updated information was unavailable for this printing.

Get_V86_Int_Vector, Get_PM_Int_Vector

Description These services return the current VM's interrupt vector for the mode specified. For V86 mode, this is the DWORD located in the real mode interrupt vector. A PM interrupt vector table is maintained by the VMM for every virtual machine.

Notice that for PM interrupts, a return value of zero indicates that the interrupt vector has not been hooked. This is an optimization so that unhooked interrupts can be immediately reflected to V86 mode without any processing in protected mode. If a protected mode application or VxD calls the DOS **Get_Vector** service, then the DOS API mapper will allocate a PM callback break point and set the appropriate interrupt vector if the vector is currently zero. The break point will invoke code that reflects the interrupt to V86 mode. Therefore, VxDs should use this service instead of the DOS **Get_vector** interface to get the current PM interrupt vector.

Entry **EAX** = Interrupt number

Exit

```
If interrupt vector points to 0:0 then
        Zero flag set
        ECX = 0
        EDX = 0
else
        Zero flag clear
        CX  = CS of vector (high word zero)
        EDX = EIP of interrupt vector (for V86 mode and 16-bit
                  protected mode programs the high word will be zero)
```

Uses **ECX**, **EDX**, Flags

Hook_V86_Int_Chain (Initialization only)

Description These services are used to monitor software interrupts and simulated hardware interrupts in Virtual 8086. More than one VxD is allowed to hook an interrupt. The last interrupt hook will be the first one called. Every interrupt hook can either service the interrupt or allow the interrupt to be reflected to the next handler in the chain. If no interrupt hook procedure consumes the interrupt, then it will be reflected to the virtual machine.

To consume an interrupt, a hook procedure must return with the Carry flag clear. If the Carry flag is set when an interrupt hook returns, then the interrupt will be passed on to the next handler in the chain or, if the end of the chain is reached, reflected to the current virtual machine.

If a VxD calls the **Simulate_Int** service, then all interrupt chain hooks will be called before the interrupt is reflected into the virtual machine. Simulated hardware interrupts will also be routed through the interrupt hooks. Therefore, your code should not assume that the VM has just executed a software interrupt instruction.

Example Windows running in enhanced mode supports an API using software interrupt 2FH. The code to handle the Release Time-Slice API looks like this:

```
Win386_Partial_API_initialization:
        mov      eax, 2fh
        mov      esi, OFFSET32 Win386_Partial_API_Hook
        VMMcall  Hook_V86_Int_Chain
        clc
        ret

Win386_Partial_API_Hook
        cmp      [epb.Client_AX], 1680h
        je       SHORT Win386_PA_Our_Call
        stc
        ret
Win386_PA_Our_Call:
        VMMcall  Release_Time_Slice
        clc
        ret
```

When **Win386_Partial_API_Hook** is called, it checks for 1680H in the VM's **AX** register. If **Client_AX** !=1680H, then it reutrns with Carry set, and the interrupt will be reflected to the next handler in the interrupt chain. However, if **Client_AX** = 1680H, then it releases the current virtual machine's time-slice and consumes the interrupt by returning with Carry clear.

Entry

EAX = Interrupt #
ESI Procedure to call

Exit

```
If carry set then
        ERROR: Invalid interrupt number
else
        Interrupt hook installed
```

Uses

Flags

Callback

EAX = Interrupt #
EBX = Current VM handle
EBP -> Client register structure

```
If the callback procedure returns with carry clear then
        The interrupt is NOT passed to the next interrupt hook
else (if carry set)
        The interrupt IS passed to the next interrupt hook
```

Set_PM_Int_Type

NOTE The description for this service has been identified as out of date and the updated information was unavailable for this printing.

Set_V86_Int_Vector, Set_PM_Int_Vector

Description This service sets the current interrupt vector for the mode specified. If a VxD calls **Set_xxx_Int_Vector** before the Sys_VM_Int control call is made, then the installed handler will become part of the default interrupt vector table. In other words, every VM will be created with interrupt vectors set during enhanced Windows environment initialization. If this service is called after **Sys_VM_Init**, then the handler will only be installed in the current virtual machine.

Entry
EAX = Interrupt number
CX = CS to set into vector
EDX = EIP to set interrupt vector (for V86 mode and 16-bit protected mode programs the high word should be zero)

Exit None

Uses Flags

Simulate_Far_Call

Description This service places the current VM's **CS:IP** on the VM's stack and puts the **CS:IP** specified in **CX:EDX** in the **Client_CS:EIP**. The next time the VM is executed, it will be as if a FAR call had been inserted in the VM's instruction stream.

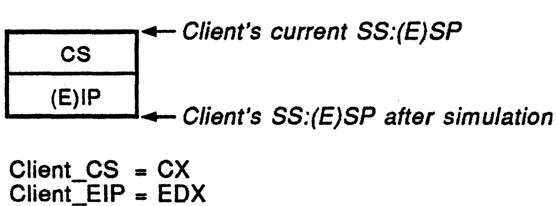


Figure 21.1 Simulate_Far_Call (?) SERV_02.EPS

Entry CX = Segment of procedure to call
EDX = Offset of procedure to call (high word 0 if 16-bit application)

Exit Old **Client_CS**, **Client_EIP**, **Client_ESP**, Flags

Simulate_Far_Jmp

Description This service places the specified **CS:IP** into the VM's **CS:IP** to simulate a FAR **jmp** instruction.

Entry **CX = CS to jump to**
EDX = **EIP** to jump to (High word should be zero for 16-bit or V86 apps)

Exit None

Uses **Client_EIP**, **Client_ESP**, Flags

Simulate_Far_Ret

Decsription This procedure pops the top two WORDs orDWORDs on the current VM's stack into the client's **CS:(E)IP**.

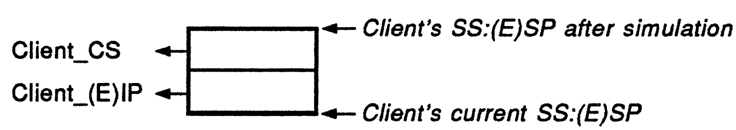


Figure 21.2 Simulate_Far_Ret (?) SERV_03.EPS

Entry None

Exit None

Uses Flags

Simulate_Far_Ret_N

Description This procedure pops the top two WORDs or DWORDs on the current VM's stack into the client's **CS:(E)IP** and, then, subtracts **EAX** from the VM's stack pointer.

Entry EAX = Number of bytes to pop after far ret

Exit None

Uses **Client_CS, Client_EIP, Client_ESP,** Flags

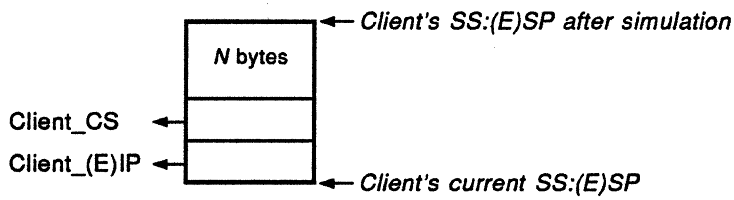


Figure 21.3 Simulate_Far_Ret_N (?) SERV_04.EPS

Simulate_Int

Description This service is used mainly by the Virtual Programmable Interrupt Controller Device to simulate hardware interrupts. Most VxD writers will want to use the **Exec_Int** service to simulate interrupts.

This service has exactly the same effect as a VM executing an **Int nn** instruction. All VxD interrupt chain hooks are called and, if the interrupt is not consumed by one of these hooks, an IRET frame is built on the VM's stack. Notice, however, that the VM interrupt code will not be executed until the enhanced Windows environment returns to the virtual machine. If you want to *execute* an interrupt, then you should use the nested execution services (**Exec_Int**).

This service is mode sensitive. Therefore, if the VM is currently in V86 mode, then a V86 interrupt will be simulated. Otherwise, a PM interrupt will be simulated. Since reflecting a PM interrupt may force a mode change to V86 mode, VxD writers must be very careful when calling this service while running a protected-mode application.

Entry EAX = Interrupt number

Exit If **Simulate_Int** is called while running a PM application and the PM interrupt vector is 0, then the mode is chnaged to V86.

Uses **Client_CS, Client_EIP, Client_Flags,** Flags

Simulate_Iret

Description This service pops the values at the top of the current VM's stack into the current VM's **CS:IP** and flags. If the current VM is a 32-bit protected-mode application, then this service will pop three DWORDs instead of WORDs (simulate an IRETD).

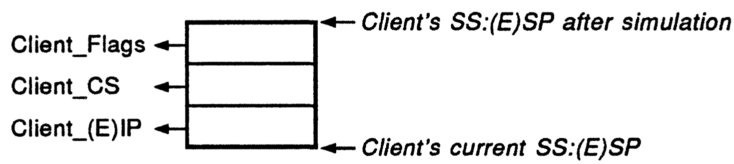


Figure 21.4 Simulate_Iret (?) SERV_01.EPS

Entry None

Exit None

Uses **Client_CS, Client_EIP, Client_ESP, Client_Flags,** Flags

Chapter 22 Nested Execution Services

These services provide a way for VxDs to call routines in a VM. Notice that the VxD must make sure that the service being called is in a callable state (i.e., you must not reenter services that do not expect to be reentered).

Begin_Nest_Exec

Description This service is used by devices that need to call software in a virtual machine. For example:

```
VMMcall Begin_Nest_Exec                    ; Start nested execution
        mov     [ebp.Client_AH], 30h       ; 30h = Get MS-DOS Version #
        mov     eax, 21h                   ; Execute an Int 21h in the
        VMMcall Exec_Int                   ; current VM to call DOS
VMMcall End_Nest_Exec                      ; End of nested exec calls
```

will make the DOS Get Version call. The version will be in the **Client_AH** and **Client_AL** registers.

This service only works for the current VM. The VM registers changed by the call WILL BE CHANGED IN THE VM. If you want to save and restore a VM's registers you should use the "**Save_Client_ State**" and "**Restore_Client_State**" services or the "**Push_Client_State**" and "**Pop_Client_State** macros."

You may execute any number of interrupts between a **Begin/End_Nest_Exec** pair. For example the following is valid:

```
VMMcall Begin_Nest_Exec
...
VMMcall Exec_Int
...
VMMcall Exec_Int
...
VMMcall Simulate_Far_Call
VMMcall Resume_Exec
...
VMMcall Exec_Int
VMMcall End_Nest_Exec
```

This service will force the VM into protected mode execution if there is a protected mode application running in the current VM. If there is no protected mode application, then the VM will remain in V86 mode. When **End_Nest_Exec** is called the VM will be returned to whatever mode it was in when **Begin_Nest_Exec** was called. For more information on what is entailed in a mode switch refer to the documentation for "**Set_PM_Exec_Mode**" and "**Set_V86_Exec_Mode**".

If the execution mode changes from V86 to PM then this service will automatically switch the VM to the locked PM stack (and **End_Nest_Exec** will switch it back). This allows most devices to change execution modes without worrying about demand paging issues.

Entry None

Exit **Client_CS:IP** contains a break point (used by nested exec services) If a protected mode application is running then
   VM execution mode is protected mode
else
   VM execution mode is Virtual 8086 mode
Exec_Int and **Resume_Exec** services may be called.

Uses **Client_CS**, **Client_IP**, Flags

Begin_Nest_V86_Exec

Description This service will set the the current VM in Virtual 8086 mode and prepare the VM for nested execution. This service is normally used by devices that want to convert protected mode calls into V86 calls. For example, the DOSMGR device uses this call to map INT 21H DOS calls issued from protected mode programs into Virtual 8086 mode DOS calls.

This call, like **Begin_Nest_Exec**, saves the current execution mode of the virtual machine (either V86 or PM) and **End_Nest_Exec** will restore the mode.

Entry None

Exit **Client_CS:IP** contains a break point (used by nested exec services)
VM is in Virtual 8086 mode.
Exec_Int and **Resume_Exec** services may be called.

USES **Client_CS**, **Client_IP**, Flags

Begin_PM_Exec

ED. NOTE Please be advised that this service may no longer be supported or may have changed. Presented here is the most current documentation available at time of printing.

Description

This service is used by devices that load protected mode applications to set the execution mode to protected mode. It will set the **VMStat_PM_App** and **VMStat_PM_Exec** status flags in the current VM's control block status field and set the current execution mode to protected mode. If the 32-bit option is selected it will also set the **VMStat_PM_Use32** flag.

It is up to the caller to save the current client registers and restore them after calling **End_PM_Exec**. None of the protected mode registers will be initialized by this call. Therefore it is up to the caller to initialize **DS, ES, FS, GS, CS, EIP, SS,** and **ESP**. Also note that the loader must allocate any memory and selectors the protected mode program will use. The loader must supply a stack segment for the application.

Typically, loaders have the following logic:

```
 Start_Load:
        mov     edi, (Per-VM buffer to save state)
        VMMcall Save_Client_State
        mov     eax, (0 or 1)
        VMMcall Begin_PM_Exec
        jc      Error
        (Load application code and data)
        (Set Client_CS:EIP to application entry point)
        (Set Client_SS:ESP to application stack stack segment)
        (Set initial values for Client_DS, ES, FS and GS)
        ret (This will jump to programs entry CS:EIP.)

End_Program: (Normally catch Int 21h, AH=4Ch)
        VMMcall End_PM_Exec
        mov     esi, (Per-VM buffer of saved client state)
        VMMcall Restore_Client_State
        ret (Returns to previous program in this VM)
```

Since more than one protected mode program may be loaded in a VM this service maintains a count. The first time it is called it sets the **VMStat_PM_App** flag and sets the execution mode to proteced mode. Subsequent calls to this will increment the counter (unless the service fails) and set the execution mode to protected mode. You must call **End_PM_Exec** once for every call to **Begin_PM_Exec**.

Entry

EAX = Flags
    Bit 0 = 1 if application is 32-bit, (0 if 16-bit)
    all other flags reserved and must be 0

Exit

```
If carry flag clear then
  Successful - VM is in PM execution mode.
```

```
  VMStat_PM_App flag set in current VM's control block status flags

else

  ERROR: Could not begin PM execution because out of memory or
         another PM application is different mode (16-bit requested
         while 32-bit running or 32-bit requested while 16-bit
         running)
```

Uses Flags

Begin_Use_Locked_PM_Stack

Description This service is used by devices that need to ensure that a protected mode program is running on a stack that will not be demand paged. Most devices can rely on **Begin_Nest_Exec** to switch stacks automatically and so this service is only important for devices such as the Virtual Programmable Interrupt Controller Device (VPICD) which explicitly change the execution mode of a VM.

A call to this service must be followed by a call to **End_Use_Locked_PM_Stack.** Note that this service may be called repeatedly, but only the first call will switch stacks. Subsequent calls will increment a counter but remain on the current locked stack.

Entry Current execution mode of VM must be protected mode (**VMStat_PM_Exec** status bit must be set).

Exit

```
If locked stack not already in use then
    Client's SS:SP will be changed to locked protected mode stack
else
    Client's SS:SP will be unchanged
```

Uses Flags

End_Nest_Exec

Description This must be called after a call to **Begin_Nest_Exec**. A device must never return to the VMM while still in nested execution. If **Begin_Nest_Exec** changed the execution mode of the VM then this service will restore it to the previous mode. Note that this service WILL NOT restore the client's registers (except **CS:IP**) to the values they were when **Begin_Nest_Exec** was called. If you need to preserve the VM's registers you must use the **Push/Pop_Client_State** macros.

Entry None

Exit VM execution mode restored to previous execution mode (before **Begin_Nest_Exec** was called)
Client's original **CS:IP** restored

Uses **Client_CS**, **Client_IP**, Flags

End_PM_ExecED

NOTE Please be advised that this service may no longer be supported or may have changed. Presented here is the most current documentation available at time of printing.

Description This service must be called once for every call to **Begin_PM_Exec**. If the internal count maintined by **Begin/End_PM_Exec** is decremented to zero then the **VMStat_PM_App** flag in the control block is cleared and and the VM will be placed in V86 execution mode. Otherwise, if the count remains greater than zero then the VM execution mode is not changed and none of the client registers will be altered.

Entry None

Exit VM may be in V86 execution mode if final **End_PM_Exec**

Uses Flags

End_Use_Locked_PM_Stack

Description This service must be called once for every call made to **Begin_Use_Locked_PM_Stack.** It will decrement the locked stack use counter and if it is decremented to zero then it will switch the VM back to it's original **SS:SP**.

Entry None

Exit

```
If locked stack count decremented to 0 then
    Client's SS:SP will be restored to original values before
    Begin_Use_Locked_PM_Stack was called.
else
    Client's SS:SP will be unchanged
```

Uses Flags.

Exec_Int

Description

YOU MUST CALL **BEGIN_NEST_EXEC** OR **BEGIN_NEST_V86_EXEC** BEFORE CALLING THIS SERVICE. IT MAY BE CALLED ANY NUMBER OF TIMES BETWEEN A BEGIN/ END NEST EXEC PAIR.

This service simulates an interrupt and then resumes VM execution. It has exactly the same effect as calling:

```
mov     eax, (Int #)
VMMcall Simulate_Int
VMMcall Resume_Exec
```

Since most nested execution calls simulate interrupts, this service is provided for convienence. See **Resume_Exec** for more details on how this service is used.

Entry

EAX = # of interrupt to execute

Exit

Interrupt has been executed

Uses

Flags

Exec_VxD_Int

Description

This service is used by virtual devices to call DOS or BIOS services as though they were an application program. For example, the following code gets the current DOS version:

```
mov     ax, 3000h
push    DWORD PTR 21h
VMMcall Exec_VxD_Int
(AL = Major DOS version, AL = Minor DOS version)
```

All DOS and BIOS calls that are supported in protected mode programs will be supported by this service. The VM's registers and flags will not be changed by this serivce so there is no need for the caller to save and restore the client register structure. The interrupt number on the stack will be removed by this serivce so the caller should NOT add four to **ESP** after calling this serivce.

To make calling this service easier, a macro called VxDint is defined in VMM.INC as follows:

```
VxDint  MACRO   Int_Number
        push    Int_Number
        VMMcall Exec_VxD_Int
        ENDM
```

This service makes it possible to write code in a virtual device that is very similar to real mode code. For example, below is the code that opens a file named "FOO.TXT" and reads the first 100 bytes:

```
VxD_DATA_SEG
        Foo_File_Name   db      "FOO.TXT", 0
        Read_Buffer     db      100 dup (?)
VxD_DATA_ENDS

VxD_CODE_SEG
        BeginProc Sample_File_Read
                mov     ax, 3D00h                       ; Open file with handle
                mov     edx, OFFSET32 Foo_File_Name     ; DS:EDX - File name
                VxDint  21h                             ; Call DOS
                jc          Error                       ; If carry then error
                                                        ; else AX = File handle
                mov     bx, ax                          ; BX = File handle
                mov     ecx, 100                        ; Read 100 bytes
                mov     edx, OFFSET32 Read_Buffer       ; Into this buffer
                mov     ah, 3Fh                         ; DOS Read
                VxDint  21h                             ; Call DOS
                jc          Error                       ; Error if carry else
                                                        ; EAX = # bytes read
                (Do stuff with the data here)

        EndProc Sample_File_Read
VxD_CODE_ENDS
```

WARNING Interrupts will only be routed through virtual device interrupt hooks. THEY WILL BYPASS ANY HOOK THE APPLICATION HAS INSTALLED IN PROTECTED MODE. This may be a problem, for example, if an application hooks Int 21h to watch file opens and then a VxD uses this service to open a file (the application would not see the file open).

Do not change **DS** or **ES** before calling this service. You should always use the ring 0 linear address of the data instead of changing the selector value. This may require using the **_SelectorMapFlat** service to determine the base of a selector.

Do not call services that will change **DS** or **ES**. Mappers should return valid pointers without changing the segment register value, but calls that explicitly change the **DS** or **ES** selectors should never be called. For example, if a call returns a pointer in **DS:(E)DX** then this would be OK to call since the mapper would convert the ponter to use the ring 0 linear address in **EDX** without modifying **DS**. However, if a service returns a selector only then you should not use **Exec_VxD_Int** to call it. This can normally be made to work by using code similar to the following:

```
Push_Client_State
VMMcall Begin_Nest_(V86_)Exec
. . .
(Fiddle with client registers)
. . .
```

```
VMMcall Exec_Int
. . .
(Get segments/selectors)
. . .
VMMcall End_Nest_Exec
Pop_Client_State
```

Entry DWORD at [ESP+4] is number of interrupt to execute

Exit All registers and flags modified by interrupt will be changed. The interrupt number on the stack will have been removed.

Uses All registers and flags modified by interrupt will be changed.

Restore_Client_State

Description This service restores a VM execution state that was saved using the **Save_Client_State** service. If the client state was saved using the **Push_Client_State** macro then you should use **Pop_Client_State** to restore the VM's execution state. The **Pop_Client_State** macro looks like:

```
Pop_Client_State MACRO
     push    esi
     lea     esi, [esp+4]
     VMMcall Restore_Client_State
     pop     esi
     add     esp, SIZE Client_Reg_Struc
     ENDM
```

Note that this service can have interesting side effects if it is not used carefully. For one thing, it will change modes from V86 to protected mode or from protected to V86 mode if the state being restored is in a different execution mode from the current one. Also, it may change the state of the current virtual machine's interrupt flag and so it may cause call-backs to events scheduled through the "**Call_When_VM_Ints_Enabled**" or "**Call_Priority_VM_Event**" services.

Entry ESI -> Buffer

Exit VM execution state is restored

Uses Flags

Resume_Exec

Description

YOU MUST CALL **BEGIN_NEST_EXEC** OR **BEGIN_NEST_V86_EXEC** BEFORE CALLING THIS SERVICE. IT MAY BE CALLED ANY NUMBER OF TIMES BETWEEN A BEGIN/ END NEST EXEC PAIR.

This service immediately executes the current virtual machine. When the virtual machine returns to the same point it was at when **Begin_ Nest_Exec** was called, this service will return. For example:

```
Push_Client_State
VMMcall  Begin_Nest_Exec
        mov      cx, [Target_CS]
        mov      eax, [Target_CS_EIP]
        VMMcall  Simulate_Far_Call
        VMMcall Resume_Exec
          (Examine results returnd in Client registers)
VMMcall End_Nest_Exec
Pop_Client_State
```

will return when the called procedure returns.The following code will process any outstanding events and immediately return:

```
VMMcall Begin_Nest_Exec
VMMcall Resume_Exec
VMMcall End_Nest_Exec
```

Since the **Resume_Exec** resumes execution at the same point that **Begin_Nest_Exec** was called it will return immediately.

This service is also useful for devices that must wait for an external event (such as a hardware interrupt) to occur before returning to the virtual machine. Since **Resume_Exec** allows outstanding events to be processed, simulated harware interrupts can be sent to the virtual machine while waiting:

```
 (Push_Client_State is not needed)
 VMMcall Begin_Nest_Exec
My_Wait_Loop:
 test    [My_Status], Done
 je      Exit_My_Wait_Loop
 VMMcall Resume_Exec
 VMMcall Release_Time_Slice
 jmp     My_Wait_Loop
Exit_My_Wait_Loop:
 VMMcall  End_Nest_Exec
 (Pop_Client_State is not needed)
```

Note that you do not need to save and restore the client registers in this loop since simulated hardware interrupts and events will not modify the client registers. You should only use the **Push/ Pop_Client_State** macros when your VxD code explicitly calls code in a virtual machine or directly modifies any client register.

This service and **Exec_Int** may be called multiple times in between calls to Begin/End nest exec. For example the following code is valid:

```
Push_Client_State
        VMMcall Begin_Nest_Exec
        mov     eax, (Int #)
        VMMcall Exec_Int
        mov     cx, [Target_CS]
        mov     eax, [Target_CS_EIP]
        VMMcall Simulate_Far_Call
        VMMcall Resume_Exec
VMMcall End_Nest_Exec
Pop_Client_State
```

Since events are processed when **Resume_Exec** (or **Exec_Int**) is called, a task switch may occur.

Entry None

Exit None

Uses Flags

Save_Client_State

Description This service will copy the contents of the current VM's Client Register Structure to the specified buffer.The buffer must be the size of the structure named "**Client_Reg_Struc**" which is defined in VMM.INC. The saved state can later be restored by calling **Restore_Client_State.**

Most of the time it is easier to use the **Push_Client_State** macro than to call this service directly. **Push_Client_State** copies the client's state onto the protected mode stack. The macro code is as follows:

```
Push_Client_State MACRO
     sub     esp, SIZE Client_Reg_Struc
     push    edi
     lea     edi, [esp+4]
     VMMcall Save_Client_State
     pop     edi
     ENDM
```

As you can see this macro will reserve space on the caller's stack for the buffer. You must use the **Pop_Client_State** macro to get rid of the contents saved on your stack. The macro will not change any registers.

This service is typically used by devices that need to make calls to code in a virtual machine that are unrelated to the current VM's thread of execution. For example, the demand paging device (PageSwap) does the following:

```
Push_Client_State
VMMcall Begin_Nest_Exec
. . .
(Perform disk I/O)
. . .
VMMcall End_Nest_Exec
Pop_Client_State
```

Note that the **Push_Client_State** macro is placed BEFORE the call to **Begin_Nest_Exec** and the **Pop_Client_State** macro is AFTER the call to **End_Nest_Exec**. Any other combination would probably crash Win386.

WARNING Always use this service to save the client state. Don't just copy the VM's client register structure and later copy it back as this will almost certianly cause Win386 to hang or crash.

Entry **EDI** -> Buffer

Exit Buffer contains a copy of the current VM's client register structure

Uses Flags

Set_PM_Exec_Mode

Description This service forces the current virtual machine into protected mode. Most devices will want to use **Begin_Nest_Exec** instead of this service.

Changing the execution mode of a VM will not change the VM's **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, and **EBP** registers or MOST flags. The VM flag and IOPL flags will change. **DS**, **ES**, **FS**, **GS**, **SS**, **ESP**, **CS**, and **EIP** will be restored to the previous values for protected mode.

If the current VM is already in protected mode then this service has no effect.

Entry None

Exit VM is in PM execution mode

Uses Flags

Set_V86_Exec_Mode

Description

This service forces the current virtual machine into V86 mode. Most devices will want to use **Begin_Nest_V86_Exec** instead of this service.

Changing the execution mode of a VM will not change the VM's **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, and **EBP** registers or MOST flags. The VM flag and IOPL flags will change. **DS**, **ES**, **FS**, **GS**, **SS**, **ESP**, **CS**, and **EIP** will be restored to the previous values for V86 mode. VM execution mode will be restored to previous execution mode (before **Begin_Nest_Exec** was called). Client's original **CS:IP** will be restored.

If the current VM is already in V86 mode then this service has no effect.

Entry

None

Exit

VM is in V86 execution mode

Uses

Flags

Chapter 23 Break Point and Callback Services

The services described in this chapter are used to handle breakpoint and callback procedures.

The discussion of these services is presented in the following order:

- **Allocate_V86_Call_Back**
- **Allocate_PM_Call_Back**
- **Call_When_VM_Returns**
- **Install_V86_Break_Point**
- **Remove_V86_Break_Point**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

Allocate_V86_Call_Back, Allocate_PM_Call_Back

Description

A *V86 callback* is used to transition from V86 mode into a protected mode VxD. The callback is a SEGMENT:OFFSET that, when executed by a V86 machine, will cause a procedure in a virtual device to be called.

A *PM callback* is used to transition from a protected-mode application to a VxD. The callback is a SELECTOR:OFFSET that, when executed, will cause a procedure in a virtual device to be called.

These services are typically used by devices that need to be called by software running in a virtual machine. When the VM software calls the callback address, the VxD gets control and can service the VM's request.

```
Initialization:
        mov     edx, My_Ref_Data
        mov     esi, OFFSET33 My_API_Procedure
        VMMcall Allocate_V86_Call_Back
        mov     [My_V86_Call_Back], eax
        mov     [ebp.Client_DI], ax
        shr     eax, 16
        mov     [ebp.Client_ES], ax
        ret
```

```
My_API_Procedure:
        . . . (Do something here) . . .
        VMMcall Simulate_Far_Ret
        ret
```

Entry

EDX = Reference data (any DWORD)
ESI = Procedure to call

Exit

EAX = CS:IP of V86 callback address

Uses

EAX, Flags

Callback

EBX = Current VM handle
EDX = Reference data
EBP -> Current VM's client register structure

Call_When_VM_Returns

Description

This service is normally used to watch the "back end" of a software interrupt. For example, assume that the procedure **I16_Hook** has been placed in the V86 interrupt chain (using the **Hook_V86_Int_Chain** service). If the procedure wants to look at the return value from INT 16H, it would use the following code:

```
I16_Hook:
        xor     eax, eax                    ; No time-out
        mov     esi, OFFSET32 I16_Return    ; Call this when iret executed
        VMMcall Call_When_VM_Returns        ; Hook the return
        stc                                 ; Reflect to next int handler
        ret

I16_Return:
        (Examine results of Int 16h)
        ret
```

This service actually replaces the client's **CS:IP** with a callback. Since at the point **I16_Hook** is executed the interrupt IRET frame has not yet been built on the client's stack, the callback will be pushed on the client's IRET frame. When the VM executes an IRET to return from the interrupt, the callback break point will be executed and control will be transferred to **I16_Return**. This service will take care of restoring the client's **CS:IP** registers to their original value.

Entry

EAX = Milliseconds until time-out (0 if no time-out)
If negative value then callback will be called for both time-out AND return (unless return before time-out).

EDX = Reference data
ESI = Address of procedure to call

Exit **Client_CS:EIP** replaced with callback address

Uses **Client_CS**, **Client_EIP**, Flags

Callback **EBX** = Current VM handle
EDX = Reference data
EBP -> Client register structure

```
If carry set then
    Time-out occurred before VM returned
else
    Client_CS:IP restored to original value
    VM returned and executed break point
    If time-out value specified was negative then
        If Zero flag set then
            Time-out DID occur. Second call to this callback
        else
            Time-out did not and will not occur.
```

Install_V86_Break_Point

Description This service is used to patch V86 code in a VM. It is primarily used by the DOSMGR device to place patches in the DOS BIOS. Most VxD will have no use for this service. A good example of a "typical" use for this service is the Windows/386 XMS virtual device. Since there is already a real mode XMS driver when the VxD environment starts, the virtual XMS device must place a V86 break point at the real XMS driver entry point so that it can intercept all XMS calls.

This service places a Windows/386 V86 break point instruction at the specified SEGMENT:OFFSET in the current virtual machine. V86 break points will normally be placed in global VM memory during device initialization. V86 break points must be placed only in RAM that will have a constant linear address (they cannot move or be placed in ROM).

When a VM executes the break point, control will be passed to the VxD that installed it. The client's (VM's) **CS:IP** will still point to the break point that caused the fault. Therefore, the virtual device must change the **CS:IP** or else the break point will be executed again when the VxD environment returns to the VM. In the case of the virtual XMS device, it would call **Simulate_Far_Ret** to return to the code that called the XMS driver. Other devices may want to simulate the instruction that was patched out and increment the **IP** past the patch, jump to another **CS:IP** using **Simulate_Far_Jmp**, or return from an interrupt handler using **Simulate_Iret**.

If a particular V86 break point is no longer needed, then the VxD should call **Remove_V86_Break_Point**. Also, any break points that are placed in global V86 code (code loaded before Windows/386 was loaded) *must* be removed at **System_Exit** time.

NOTE The segment used to install a V86 break point must be the code segment the virtual machine will use when it executes the code that is being patched. For example, if you place a patch at 0100:0010 and the virtual machine hits the break point at 00FF:0000h (which is the same linear address as 0100:0010), then an error will occur even though the VM executed a valid break point.

Entry

EAX = CS:IP
EDX = Reference data (any DWORD)
ESI = Offset of procedure to call

Exit:

```
If carry set then
    Could not install break point
 else
    V86 break point successfully installed
```

Uses

Flags

Callback

EAX = **Client CS:IP** that faulted
EBX = Handle of current VM
EDX = Reference data
ESI = Linear address of break point (CS << 4 + IP)
EBP -> Client register structure

Remove_V86_Break_Point

Description

This service is used to remove a V86 break point that was installed using the **Install_V86_Break_Point** service. It will restore the original contents of the memory automatically.

Entry

EAX = **CS:IP** of break point to remove

Exit

```
If carry set then
    ERROR: Not a valid V86 break point
 else
    Previous value restored at break point SEG:OFFSET
```

Uses

Flags

Chapter 24 Primary Scheduler Services

Each virtual machine is a separate task in the enhanced Windows environment. There are several services that are used to control the scheduling of virtual machines.

Every VM has an execution priority. The VM with the highest execution priority is allowed to run unless the VM is suspended or is blocked waiting for a critical section to be freed. A VM's execution priority can be raised or lowered using the **Adjust_Execution_Priority** service.

A VxD can force a particular virtual machine to run by boosting its execution priority. However, VxD authors should take care when changing the priority of a VM since doing so can radically effect the behavior of the Windows time-slicer.

To allow the mutual exclusion of non-reentrant code, the scheduler supports a single critical section. The current VM can claim the critical section at any time by calling **Begin_Critical_Section**. If another VM owns the critical section, then the current VM will block until the critical section is released. Once the critical section is claimed, the VM's execution priority is boosted. However, VMs with higher priorities will still be allowed to execute. Normally, VMs are only boosted higher than the critical section priority when a hardware interrupt is simulated.

A VM may be suspended if it is not in a critical section. However, the system VM can never be suspended. A suspended VM will never be scheduled, regardless of its execution priority, until it is resumed.

An important thing to keep in mind is that since the enhanced Windows environment is a single-threaded operating system, you do not have to be concerned with a task switch from within a procedure. For example, another VM will not be scheduled while in a virtual device I/O trap handler. Task switches take place when a VxD makes an explicit call to the scheduler (i.e., **End_Critical_Section**) or at event processing time. Notice that since events are processed when **Resume_Exec** or **Exec_Int** are called, a task switch may occur while performing nested VM execution. Also, touching or locking unlocked demand-paged memory may cause a task-switch. In summary, the times when a task switch may occur are as follows:

- Explicit calls to the scheduler
- Performing nested execution (**Resume_Exec** or **Exec_Int**)
- Touching or locking demand-paged memory

The discussion of services providing support for the Primary Scheduler is presented in the following order:

- **Adjust_Exec_Priority**
- **Begin_Critical_Section**
- **Call_When_Not_Critical**
- **Call_When_Task_Switched**
- **Claim_Critical_Section**
- **End_Crit_And_Suspend**
- **End_Critical_Section**
- **Get_Crit_Section_Status**
- **No_Fail_Resume_VM**
- **Nuke_VM**
- **Release_Critical_Section**
- **Resume_VM**
- **Suspend_VM**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

Adjust_Exec_Priority

Description This service is used to raise or lower the execution priority of the specified VM. Since the non-suspended VM with the highest execution priority is always the current VM, this service will cause a task switch under two circumstances:

1. The execution priority of the current VM is lowered (EAX is negative), and there is another VM with a higher priority that is not suspended.
2. The execution priority of a non-suspended VM which is not the current VM is raised (EAX is positive) higher than the current VM's execution priority.

Note that even if the current VM is in a critical section, a task switch will still occur if the priority of another non-suspended VM is raised higher than the current VM's priority. However, this will only happen when a VM is given a time-critical boost, for example, to simulate a hardware interrupt. There are equates defined in VMM.INC that should be used when adjusting a VM's priority. They are listed below in order from lowest to highest.

Equate Name	Description
Reserved_Low_Boost	Reserved for use by system.
Cur_Run_VM_Boost	Time-slice scheduler boosts each VM in turn by this value to force them to run for their alloted time-slice.
Low_Pri_Device_Boost	Used by VxDs that need an event to be processed in a timely fashion but that are not extremely time critical.
High_Pri_Device_Boost	Time critical operations that should not circumvent the critical section boost should use this boost.
Critical_Section_Boost	VM priority is boosted by this value when **Begin_Critical_Section** is called.
Time_Critical_Boost	Events that must be processed even when another VM is in a critical section should use this boost. For example, VPICD uses this when simulating hardware interrupts.
Reserved_High_Boost	Reserved for use by system.

It is often more convienient to call **Call_Priority_VM_Event** than to call this service directly.

Entry

EAX = + or - priority boost (signed long integer)
EBX = VM handle

Exit

None

Uses

Flags

Begin_Critical_Section

Description

Use of this service causes the current VM to enter a global critical section. Only one VM can own the critical section at a time. If a VM calls this service while another VM owns the critical section, then the current VM will block until the critical section is released.

The critical section is maintained as a count and so *n* calls to **Begin_Critical_Section** must be followed by *n* calls to **End_Critical_Section** before the VM will leave the critical section.

When the critical section is first claimed, the execution priority of the current VM is boosted by the **Critical_Section_Boost** value defined in VMM.INC. This means that task switches to other VMs will only occur for time-critical operations such as simulating hardware interrupts.

Critical sections are used for code that must not be entered in more than one VM. For example, while in DOS, the DOSMGR VxD places the VM in a critical section. If another VM makes a DOS call, then it will block until the critical section owner's DOS call completes. However, this scenario is unlikely since a VM has an extremely high execution priority while it owns the critical section, and, therefore, other VMs will not run until the critical section is released. A scenario that *would* cause a VM to block is as follows:

```
VM X calls DOS to read a file.
The DOSMGR calls Begin_Critical_Section for VM X. This raises
     VM X's priority by the Critical_Section_Boost.
The Virtual Keyboard Device simulates an interrupt to VM Y.
VM Y is sceduled since it has a higher execution priority
     (simulated interrupts use the Time_Critical_Boost).
A T&SR program "wakes up" on the keyboard interrupt and calls DOS.
The DOSMGR calls Begin_Critical_Section for VM Y.
VM Y blocks since another VM owns the critical section.
VM X is scheduled since it has the highest exectution priority.
The DOS read for VM X completes.
DOSMGR calls End_Critical_Section for VM X.  This lowers
     VM X's priority by the Critical_Section_Boost.
VM Y is un-blocked and scheduled since it has the highest priority.
VM Y continues execution at the instruction immediately after the
     call to Begin_Critical_Section and executes the DOS call.
```

Sometimes it is preferable to boost the current VM by the **Time_Critical_Boost** value instead of entering a critical section. This prevents the main thread of execution from running in all but the current VM but avoids blocking a VM when it is not really necessary.

Entry None

Exit None

Uses Flags

Call_When_Not_Critical

Description This service will call a VxD when the critical section is released. Notice that it will not execute the callback until the current VM's execution priority is less than the **Critical_Section_Boost** even when the current VM is *not* in a critical section. This is done because most VxDs that use this service will want to wait until the critical section is free and no hardware interrupts are being simulated.

Normally it is more convenient to use the **Call_Priority_VM_Event** service than to call this service directly.

Entry **ESI** = Address of call-back procedure
EDX = Reference data to pass to callback procedure

Exit None

Uses Flags

Callback **EBX** = Current VM handle
EDX = Reference data
EBP -> Client register structure

Procedure can corrupt **EAX, EBX, ECX, EDX, ESI, EDI**, and Flags

Call_When_Task_Switched

Description This service provides a way to be informed each time a different VM is to be executed. The specified procedure will be called *every time* a task switch occurs. Since this is a frequent operation in most environments, this service should be used sparingly, and the callback procedure should be optimized for speed.

VxDs must sometimes save the state of a hardware device every time a task switch occurs and restore the hardware state for the VM that is about to be run. However, VM events can often be used in place of using this service.

Entry **ESI** = Pointer to procedure to call at task switch time

Exit None

Uses Flags

Callback **EAX** = Handle of VM switching away from (old **Cur_VM_Handle**)
EBX = Current VM (just switched to)

Procedure can destroy **EAX, EBX, ECX, EDX, ESI, EDI** and Flags

Claim_Critical_Section

Description This service will increment the critical section count by the specified value. It has the same effect as calling **Begin_Critical_Section** repeatedly but is faster. Refer to the documentation for **Begin_Critical_Section** for more information on the various side effects of entering a critical section.

Entry ECX = # of times to claim the critical section (0 is valid & ignored)

Exit None

Uses Flags

End_Crit_And_Suspend

Description This service will release the critical section and immediately suspend the current VM. It is used to block a VM until another event can be processed. This service is used by the Shell VxD to display Windows dialog boxes using code similar to this:

```
Show_Dialog_Box:
        VMMcall         Get_Crit_Section_Status
        jc              Cant_Do_It!
        VMMcall         Begin_Critical_Section
        mov             eax, Low_Pri_Device_Boost
        VMMcall         Get_Sys_VM_Handle
        mov             ecx, 11b
        mov             edx, OFFSET32 (Dialog_Box_Data_Structure)
        mov             esi, OFFSET32 Show_Dialog_Event
        VMMcall         Call_Priority_VM_Event
        VMMcall         End_Crit_And_Suspend
        jc              Did_Not_Work!
                        ; (When End_Crit_And_Suspend returns the dialog box
                        ; will have been displayed)

Show_Dialog_Event:
        (Call Windows to display the dialog box)
        mov     ebx, [Handle_Of_VM_That_Called_Show_Dialog_Box]
        VMMcall Resume_VM
        jc       Error!
        ret
```

The **Show_Dialog_Box** procedure enters a critical section to prevent the **Call_Priority_VM_Event** service from switching to the system VM immediately. It then calls **End_Crit_And_Suspend**, which blocks the current VM. The **Show_Dialog_Event** procedure runs in the system (Windows) VM and actually displays the dialog box. When it is finished, it resumes the VM that called **Show_Dialog_Box**.

This service must only be called when the critical section has been claimed *once*. That is the reason for the initial test of the critical section state in the **Show_Dialog_Box** procedure in the sample code.

Entry None

Exit

```
If carry set then
    ERROR: Could not suspend VM or could not release critical
    section (crit claim count != 1)
else
    Call worked. VM execution restarted by another VM calling
    "Resume_VM".
```

Uses Flags

End_Critical_Section

Description This service is used to release the global critical section after a call to **Begin_Critical_Section** has been issued. If the critical section ownership count is decremented to 0, then ownership of the critical section is released. Since releasing the critical section lowers the execution priority of the current VM, this service will cause a task switch if a non-suspended VM has a higher priority.

Entry None

Exit None

Uses Flags

Get_Crit_Section_Status

Description This service returns the critical section claim count in **ECX** and the owner of the critical section in **EBX**. If **ECX** is 0, then the current VM handle will be returned in **EBX.**

If this service returns with the Carry flag set, then the VM is in a time-critical operation such as a hardware interrupt simulation. (It has an execution priority = **Critical_Section_Boost.**)

Entry None

Exit

EBX = VM handle of current owner (Current VM if **ECX** = 0)
ECX = # of times critical section claimed
If carry set then VM is in a time-critical operation or critical section.

Uses Flags

No_Fail_Resume_VM

NOTE The description for this service has been identified as out of date and the updated information was unavailable for this release.

Nuke_VM

Description This service is used to close a VM that has not yet terminated normally. It is usually called by the Shell VxD to close VMs that the user has selected to terminate using the Window Close option on the VM's system menu.

Needless to say, this service should be used very cautiously.

Entry **EBX** = Handle of VM to destroy

Exit

```
If entry EBX = Current VM handle then
     This service will never return (same as Crash_Cur_VM)
 else
     If EBX = System VM handle then
        This service will never return (fatal error-crash to DOS)
     else
        VM has been nuked
```

Uses Flags

Release_Critical_Section

Description This service will decrement the critical section count by the specified value. It has the same effect as calling **End_Critical_Section** repeatedly but is faster.

Entry **ECX** = # of times to release ownership of critical section (0 valid)

Exit None

Uses Flags

Resume_VM

Description

This service is used to resume the execution of a VM that was previously suspended by a call to **Suspend_VM.** If the suspend count is decremented to 0, the VM will be placed on the queue of ready processes. A task switch will occur to the resumed VM if it has a higher priority than the current VM.

It is sometimes not possible to resume a VM. Normally, this is because a VxD is unable to lock the VM's memory handles. Every VxD is notified when a VM is resumed and can fail the call. In this case, this service will return with Carry set, and the VM will remain suspended with a suspend count of 1.

Entry

EBX = VM handle

Exit

```
If carry clear then
    If suspend count decremented to 0 then VM is runnable
 else
    Error could not resume (Suspend count remains 1)
```

Uses

Flags

Suspend_VM

Description

This service will suspend the execution of a specified Virtual Machine. Any VM, except the system VM, that is not in a critical section can be suspended. This service will fail if the specified VM is the critical section owner or the system VM. The system VM can never be suspended.

This service maintains a count that is incremented each time a VM is suspended. Therefore, if this service is called *n* times for a given VM, **Resume_VM** must be called *n* times before the VM will be executed.

When a VM is being suspended for the first time (its suspend count is incremented from 0 to 1), all devices will receive a control call with **EAX = VM_Suspend**. Devices may *not* refuse to suspend a VM. However, VxDs are allowed to fail the **VM_Resume** control call. Subsequent calls to **Suspend_VM** will not result in a **VM_Suspend** control call until the VM has been resumed.

When a VM is suspended, the **CB_VM_Status** field in the control block will have the **VMStat_Suspended** bit set. When a VM is suspended, VxDs should not touch any memory owned by that VM unless the VxD has previously locked the memory. You *may,* however, examine or modify the contents of a suspended VM's control block.

Entry

EBX = VM handle

Exit

```
If carry flag clear then
    VM suspended
 else
    Error: Could not suspend VM (VM is in a critical section or
    is the system VM)
```

Uses

Flags

Chapter 25 Time-Slice Scheduler Services

The enhanced Windows time-slice scheduler is the preemptive multitasking portion of the scheduler. It relies on time-slice priorities and flags to determine how much CPU time should be allocated to various virtual machines.

Every VM has a foreground (focus) and a background time-slice priority. These should be distinguished from a VM's Execution Priority. A VM with the largest Execution Priority will run, preventing other VMs from executing. The VM with the largest time-slice priority will run more often than other VMs but it will not necessarily prevent other VMs from executing.

There are three flags that affect the way the time-slicer schedules virtual machines: **VMStat_Exclusive**, **VMStat_Background**, and **VMStat_High_Pri_Background.** These flags are saved in the **CB_VM_Status** field of each VM's control block. You may examine these flags but you must never modify them directly. To change any of the flags, you must call the **Set_Time_Slice_Priority** service.

If a VM that has the **VMStat_Exclusive** bit set is assigned the execution focus, then it will become the only VM that is allowed to run. In this case, foreground and background priorities are meaningless since the VM is using 100 percent of the CPU time. The **Release_Time_Slice** service has no effect on an exclusive virtual machine. High-priority background VMs will not run when an exclusive VM has the execution focus.

If the VM with the focus is *not* exclusive, then any VM that has the **VMStat_Background** flag set will be allowed to run based on their background time-slice priority. The VM with the focus will be scheduled based on its foreground time-slice priority.

For this scheduler, a higher priority indicates that the VM should get *more* CPU time. The larger the priority, the faster the VM will run.

The algorithm used to allocate time determines the percentage of CPU time each VM should get based on their percentage of the total of all the time-slice priorities. For example, assume the following VMs exist:

VM	Foreground Priority	Background Priority	Flags
1	100	50	Exclusive, Background
2	100	50	Background
3	50	25	(none – foreground, non-exclusive)
4	250	75	Background

If the execution focus is set to VM 1, then it will use 100 percent of the CPU time since it has the exclusive flag set. If the execution focus is set to VM 2, then VMs 1, 2, and 4 will run.VM 3 would not be scheduled since it does not have the background flag set.

To determine how much time each VM should be allocated, the time-slicer first sums all the VM priorities and, then, calculates the percentage of CPU time each VM should receive as follows:

```
VM 2 foreground pri  = 100 / 225 * 100 = 45% of CPU
VM 1 background pri =  50 / 225 * 100 = 22% of CPU
VM 4 background pri =  75 / 225 * 100 = 33% of CPU
                       ---
              Total   225
```

Notice that a foreground priority of 10,000 (the maximum allowed) is special. When a VM with priority 10,000 is the execution focus VM, only high-priority background VMs will run unless the focus VM explicitly releases its time slice. This is different from an exclusive VM since other VMs *can* run if the focus gives up its time.

High-priority background VMs execute when a priority 10,000 VM has the focus even if the focus VM is not releasing its time.

The discussion of services providing support for the Time-Slice Scheduler is presented in the following order:

- **Adjust_Execution_Time**
- **Get_Execution_Focus**
- **Get_Time_Slice_Granularity**
- **Get_Time_Slice_Priority**
- **Release_Time_Slice**
- **Set_Execution_Focus**
- **Set_Time_Slice_Granularity**
- **Set_Time_Slice_Priority**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

Adjust_Execution_Time

Description

This service allows a device to change the amount of time a VM will be allowed to execute regardless of the VM's time-slice priority. Usually this service is used by devices such as the Virtual COM Device to boost temporarily the priority of a VM that is receiving lots of interrupts. This service can also be used to reduce the amount of time a VM will be allowed to run by passing a negative value in **EAX**. However, this is likely to cause execution starvation and is discouraged.

The value specified in **EAX** is the number of additional (or fewer) milliseconds the VM will be allowed to run. It has the same effect on all VMs regardless of their time-slice priority. This means that if a VxD calls this service with **EAX** = 1000, then the specified VM will be allowed to run an additional second regardless of its time-slice priority.

Notice that if the specified VM is not on the time-slice execution list, then this service will do nothing. It will *not* force a non-runnable VM to execute. In other words, a non-background VM cannot be forced to run in the background by boosting its execution time.

Be careful not to abuse this service! It can result in starvation for other processes.

Entry

EAX = + or - milliseconds to adjust execution time by
EBX = VM handle

Exit

None

Uses

Flags

Get_Execution_Focus

Description

This service returns the handle of the VM that is the focus or foreground VM. This service can be called from an interrupt handler.

Entry

None

Exit

EBX = Handle of VM with execution focus

Uses

EBX, Flags

Get_Time_Slice_Granularity

Description

This service returns the current time-slice granularity in **EAX**. The value returned is the minimum number of milliseconds a VM will be allowed to run before being rescheduled.

Entry	None
Exit	EAX = Minimum time-slice size in milliseconds
Uses	EAX, Flags

Get_Time_Slice_Priority

Description This service returns the time-slice execution flags, the foreground and background priorities, and the percent of CPU usage for a specified VM. Notice that the percent of CPU time returned indicates the amount of time the VM is allowed to run, but this number will not reflect the actual amount of CPU time if any VM releases its time slice since other VMs will be allowed to execute during that VM's time slice.

Entry EBX = VM handle

Exit EAX = Flags (Appropriate flags from CB_VM_Status control block field)
VMMStat_Exclusive
VMStat_Background
VMStat_High_Pri_Background
ECX = Foreground time-slice priority (high word 0)
EDX = Background time-slice priority (high word 0)
ESI = % of total CPU time used by VM

Uses Flags

Release_Time_Slice

Description This service causes the current VM to give up any time remaining in its current time slice and allows the next VM in the time-slice queue to run. This service should be called whenever a VM is idle to allow other VMs to execute faster. If there is only one VM in the time-slice queue, this service will do nothing.

Entry None

Exit None

Uses Flags

Set_Execution_Focus

Description This service changes the time-slice exection focus to the specified virtual machine. The VM with the focus executes with its foreground priority. If the **VMStat_Exclusive** flag is set, then it will be the only VM scheduled. Otherwise, background VMs will be allowed to run. All VMs except the focus VM, background VMs, and the system VM will be suspended.

Entry **EBX** = VM handle

Exit None

Uses Flags

Set_Time_Slice_Granularity

Description This service is used to change the minimum amount of time the time-slice scheduler will allocate to a VM. Smaller values will make multitasking appear smoother but will increase overhead due to the large number of task switches required. Larger values will allow more time for the VMs to execute but may make execution appear sporadic to the user.

Entry **EAX** = Minimum time-slice size in milliseconds

Exit None

Uses Flags

Set_Time_Slice_Priority

Description This service sets the time-slice execution flags (background, high-priority background, and exclusive status flags) and the foreground and background priorities for a specified VM.

To change part of a VM's time-slice priority status, first call **Get_Time_Slice_Priority**, then change only the values you are interested in and call this service. For example, to set a VM into background mode, you would do the following:

```
mov     ebx, [Handle_Of_VM_To_Change]
VMMcall Get_Time_Slice_Priority
or      eax, VMStat_Background
VMMcall Set_Time_Slice_Priority
```

Entry

EAX = Flags

VMStat_Exclusive
VMStat_Background
VMStat_High_Pri_Background

EBX = VM handle
ECX = Foreground priority (high word must be 0)
EDX = Background priority (high word must be 0)

Exit

```
If carry set then
    ERROR: Could not change priority / flags for VM
else
    Priority and flags changed
```

Uses

Flags

Chapter 26

Event Services

Enhanced Windows is a single-threaded, non-reentrant operating environment. Because it is non-reentrant, virtual devices that hook hardware interrupts must have some method of synchronizing their calls to VMM. For this reason, enhanced Windows has the concept of "event" processing.

When a VxD is entered due to an asynchronous interrupt, such as a hardware interrupt, the device is limited to a very specific subset of functions. It is allowed to do only the following:

- Call any Virtual PIC Device (VPICD) service
- Call any asynchronous VMM service (see individual services for details)
- Schedule events

Obviously, devices that service hardware interrupts will often need to use services other than the ones listed above. When this is the case, the VxD will need to schedule an event. When an event is scheduled, the caller defines a procedure to call when it is OK to make any VMM call. When VMM calls this procedure, the VxD can finish processing the asynchronous event.

VM events are often useful for devices that do not service hardware interrupts and can be scheduled at any time except during a Non-Maskable Interrupt (NMI).

When an event service routine is called, it is entered with the following:

- **EBX** = Current VM handle
- **EDX** = Reference data passed when the routine was set up
- **EBP** -> Client register structure

The event callback procedure can modify **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, and **EDI**.

The discussion of services providing support for events is presented in the following order:

- **Call_Global_Event**
- **Call_Priority_VM_Event**
- **Call_VM_Event**

- **Cancel_Global_Event**
- **Cancel_Priority_VM_Event**
- **Cancel_VM_Event**
- **Schedule_Global_Event**
- **Schedule_VM_Event**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

Call_Global_Event

Description This procedure is a faster method of servicing asynchronous events. If the current thread of execution begins in a virtual machine (it was *not* an interrupt from within the VMM), then the event procedure will be called immediately. Otherwise, the event will be scheduled.

Entry

ESI = Offset of procedure to call
EDX = Reference data (will be passed back to procedure)

Exit

```
If ESI = 0 then
    Event procedure was called
else
    ESI = Event handle (can be used to cancel events)
```

Uses ESI, Flags

Callback

EBX = Current VM handle
EDX = Reference data
EBP -> Client register structure

Call_Priority_VM_Event

Description This service combines the functionality of **Call_VM_Event, Call_When_VM_Ints_Enabled, Call_When_Not_Critical**, and **Adjust_Exec_Priority** into one, easy to use service. As with all event services, this service can be called from an interrupt handler.

Call_Priority_VM_Event is used by VxDs for several purposes. The most common uses are as follows:

- To wait until a VM enables interrupts and the critical section is free so the VxD can call DOS or some other non-reentrant code.

- To boost a VM's priority and wait until the VM enables interrupts to simulate an interrupt type event. For example, the VNETBIOS uses this service for asynchronous network request POST callbacks.
- To force an event to be processed in another VM by boosting the VM's Execution Priority.

Example

Assume a VxD implements a print spooler that will call a VM back when a buffer has been sent to the printer. It could use this service to notify the appropriate VM that its buffer has been printed as follows:

```
VxD_Code_SEG
    BeginProc Print_Buff_Empty
        mov  eax, Low_pri_Device_boost
        mov  ebx, [Call_Back_VM_Handle]
        mov  ecx, PEF_Wait_ForSTI or PEF_Wait_Not_Crit
        mov  edx, [Call_back_CS_IP]
        mov  esi, Buff_Empty_Call_Back_Event
        VMMCall   Call_Priority_VM_Event
        ret
    EndProc  Print_Buff_Empty
    BeginProc  Buff_Empty_Call_Back_Event
        VMMcall  Begin_Next_Exec        ;Get ready to call VM
        mov   ecx,  edx
        shr   edx,  16                  ;ECX = Segment to call
        movzx  edx, dx                  ;EDX = Offset to call
        VMMcall    Build_Int_Stack_Frame
        VMMcall    Resume_Exec          ;call the VMM's
                                        ;callback ret
    EndProc    Buff_Empty_Call_Back_Event

VxD_CODE_ENDS
```

The **Print_Buff_Empty** procedure could be called from a hardware interrupt handler in any virtual machine. It uses **Call_Priority_VM_Event** to force the correct VM to be scheduled. The priority boost specified in **EAX** will force the event to be processed quickly although not as fast as a hardware interrupt. The options specified in the **ECX** register will force the event to be delayed until the critical section is free and the VM's interrupts are enabled. The reference data in **EDX** contains the **CS:IP** of the procedure to call in the VM.

When **Buff_Empty_Call_Back_Event** is called it can make several assumptions: it is running in the desired VM, the critical section is not owned, and the VM has enabled interrupts. It uses the **CS:IP** value passed in **EDX** to simulate a pseudo-interrupt in the VM. The procedure called in the VM would have to execute an IRET to return from the callback. When **Buff_Empty_Call_Back_Event** returns, the execution priority boost is automatically deducted.

THIS EXAMPLE IS INCOMPLETE! — An actual VxD handler would need to do more work. It does not address several problems. For example "Buff_Empty_Call_Back_Event" does not take into account whether the call should

be made to a V86 CS:IP or protected mode CS:IP. It also would not work for 32-bit protected mode programs since it would need to pass a 32-bit offset (EIP) to Simulate_Far_Call.

Entry

EAX = Priority boost (can be 0)
EBX = VM handle
ECX = Option flags (defined in VMM.INC)
PEF_Wait_For_STI - Event will not be called until VM enables interrupts
PEF_Wait_Not_Crit - Event will not be called until VM is not in a critical section or time-critical operation.
PEF_Dont_Unboost - Priority of VM will not be reduced after return from event procedure.
All other bits are reserved and must be 0.
EDX = Reference data (will be passed back to procedure)
ESI = Offset of procedure to call

Exit

```
If ESI = 0 then
      Event procedure already called
    else
      Event procedure will be called later
ESI = Event handle (can cancel using Cancel_Priority_VM_Event)
```

Uses

Flags

Callback

EBX = Current VM handle
EDX = Reference data
EBP -> Client register structure

Procedure can modify **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, and Flags

Call_VM_Event

Description

This procedure is a faster method of servicing asynchronous events. If the current thread of execution begins in a virtual machine (it was *not* an interrupt from within the VMM) *and* the event is for the current VM, then the event procedure will be called immediately. Otherwise, the event will be scheduled.

Entry

EBX = VM handle
ESI = Offset of procedure to call
EDX = Reference data (will be passed back to procedure)

Exit

```
If ESI = 0 then
    Event procedure was called
else
    ESI = Event handle (can be used to cancel events)
```

Uses Flags

Callback

EBX = Current VM handle
EDX = Reference data
EBP -> Client register structure

Cancel_Global_Event

Description This service is used to cancel an event that was previously scheduled by **Schedule_Global_Event** or **Call_Global_Event**. Notice that, once a scheduled event is serviced, you must not attempt to cancel that event.

NOTE It is valid to pass **ESI** = 0 to this service (it will do nothing). This is provided so that code that uses this service can use 0 to indicate no event scheduled and not have to perform a test every time it wants to cancel an event. For example:

```
xor     esi, esi
xchg    esi, [My_Event_Handle]
VMMcall Cancel_Global_Event
```

will always work even if no event was scheduled. You will also need to set **[My_Event_Handle]** to 0 in your event procedure.

Entry **ESI** = Event handle (0 is acceptable)

Exit Global event has been canceled

Uses Flags

Cancel_Priority_VM_Event

Description This service is used to cancel an event that was previously scheduled by **Call_Priority_VM_Event.** Notice that once a scheduled event is serviced, you must not attempt to cancel that event.

NOTE It is valid to pass **ESI** = 0 to this service (it will do nothing). This is provided so that code that uses this service can use 0 to indicate no event scheduled and not have to perform a test every time it wants to cancel an event. For example:

```
xor     esi, esi
xchg    esi, [My_Event_Handle]
VMMcall Cancel_VM_Event
```

will always work even if no event was scheduled. You will also need to set **[My_Event_Handle]** to 0 in your event procedure.

Do not use this service to cancel events scheduled using the **Call_VM_Event** or **Schedule_VM_Event** services. You must cancel normal VM events using the **Cancel_VM_Event** service.

Entry **ESI** = Priority event handle (0 is valid)

Exit Event canceled, **ESI** contains garbage

Uses Flags, **ESI**

Cancel_VM_Event

Description This service is used to cancel an event that was previously scheduled by **Schedule_VM_Event** or **Call_VM_Event**. Notice that, once a scheduled event is serviced, you must not attempt to cancel that event.

NOTE It is valid to pass **ESI** = 0 to this service (it will do nothing). This is provided so that code that uses this service can use 0 to indicate no event scheduled and not have to perform a test every time it wants to cancel an event. For example:

```
xor     esi, esi
xchg    esi, [My_Event_Handle]
VMMcall Cancel_VM_Event
```

will always work even if no event was scheduled. You will also need to set **[My_Event_Handle]** to 0 in your event procedure.

Do not use this service to cancel events scheduled using the **Call_Priority_VM_Event** service. You must cancel priority events using the **Cancel_Priority_VM_Event** service.

Entry **EBX** = VM handle
ESI = Event handle (0 is acceptable)

Exit None

Uses Flags

Schedule_Global_Event

Description This procedure is used to schedule asynchronous events that are not VM specific. The events will be processed immediately before the VMM IRETs to any VM.

Entry **ESI** = Offset of procedure to call
EDX = Reference data (will be passed back to procedure)

Exit **ESI** = Event handle (can be used to cancel event)

Uses **ESI**, Flags

Callback **EBX** = Current VM handle
EDX = Reference data
EBP -> Client register structure

Schedule_VM_Event

Description This procedure is used to schedule asynchronous events that are VM specific. The events will be processed immediately before the VMM IRETs to the specified VM.

VM events will only be executed in the VM for which they were scheduled for. Therefore, if a VM event is scheduled for a VM other than the current virtual machine, it will not be processed until a task switch occurs to that VM.

Entry **EBX** = VM handle
ESI = Offset of procedure to call
EDX = Reference data (will be passed back to procedure)

Exit **ESI** = Event handle (can be used to cancel event)

Uses **ESI**, Flags

Callback

EBX = Current VM handle (VM event was scheduled for)
EDX = Reference data
EBP -> Client register structure

Chapter 27 Timing Services

Timing services are provided for use by VxDs that need to perform periodic operations or need to establish the amount of time elapsed since a particular event. They are described here in the following order:

- **Cancel_Time_Out**
- **Get_Last_Updated_System_Time**
- **Get_Last_Updated_VM_Exec_Time**
- **Get_System_Time**
- **Get_VM_Exec_Time**
- **Set_Global_Time_Out**
- **Set_VM_Time_Out**
- **Update_System_Clock**

See Chapter 16, "Overview of Windows in Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

Cancel_Time_Out

Description This service is used to cancel a time-out that was scheduled through either **Set_VM_Time_Out** or **Set_Global_Time_Out.**

NOTE It is valid to pass ESI = 0 to this service (it will do nothing). This is provided so that code that uses this service can use 0 to indicate no time-out scheduled and not have to perform a test every time it wants to cancel a time-out. For example:

```
xor     esi, esi
xchg    esi, [Local_Time_Out_Handle]
call     Cancel_Time_Out
```

will always work even if no time-out was scheduled.

Entry **ESI** = Time-out handle to cancel OR 0 if no time-out to be canceled

Exit Time-out is canceled, old time-out handle now invalid

Uses Flags

Get_Last_Updated_System_Time

NOTE The description for this service has been identified as out of date and the updated information was unavailable for this release.

Get_Last_Updated_VM_Exec_Time

NOTE The description for this service has been identified as out of date and the updated information was unavailable for this release.

Get_System_Time

Description This service will return the time in milliseconds since the enhanced Windows environment was started. There is no way to detect rollover of the clock through this function but the clock will take 49.5 days to roll over.

If you are concerned about rollover, you should schedule a time-out every 30 days.

Entry None

Exit EAX = Elapsed time in milliseconds since enhanced Windows was started

Uses EAX, Flags

Get_VM_Exec_Time

Description This service returns the amount of time that a particular VM has executed. Every VM starts with an **Exec_Time** of 0 when it is created, and the **Exec_Time** is only increased when the VM is actually executed. Therefore, the value returned does *not* reflect the length of time the VM has existed. Instead, it indicates the amount of time that task has actually been the currently running VM.

Entry None

Exit **EAX** = Amount of time in milliseconds that VM has executed

Uses **EAX**, Flags

Set_Global_Time_Out

Description Schedules a time-out that will occur after **EAX** milliseconds have elapsed.

The callback procedure will be called with **ECX** equal to the number of milliseconds that have elapsed since the actual time-out occurred. Time-outs are often delayed by 10 milliseconds or more since the normal system timer runs at 20 milliseconds or slower. If you need more accurate time-outs, then you must increase the timer interrupt frequency. See the VTD documentation for more details on setting the timer interrupt period.

Entry

EAX = Number of milliseconds to wait until time-out
EDX = Reference data to return to procedure
ESI = Address of procedure to call when time-out occurs

Exit

```
If time-out was NOT scheduled then
    ESI = 0 (This is useful since 0 = NO TIME-OUT SCHEDULED)
 else
    ESI = Time-out handle (used to cancel time-out)
```

Uses **ESI**, Flags

Callback

EBX = Current VM handle
ECX = Number of EXTRA milliseconds that have elapsed
EDX = Reference data
EBP -> Client register structure
Procedure may corrupt **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, and Flags

Set_VM_Time_Out

Description Schedules a time-out that will occur after a VM has executed for the specified length of time. Notice that the time-out will occur after the VM has run for **EAX** milliseconds. Therefore, if there is more that one VM executing, it may take more than **EAX** milliseconds to occur.

The callback procedure will be called with **ECX** equal to the number of milliseconds that have elapsed since the actual time-out occurred. Time-outs are often delayed by 10 milliseconds or more since the normal system timer runs at 20 milliseconds or slower. If you need more accurate time-outs, then you must increase the timer interrupt frequency. See the VTD documentation for more details on setting the timer interrupt period.

Entry

EAX = Number of milliseconds to wait until time-out
EBX = VM handle
EDX = Reference data to return to procedure
ESI = Address of procedure to call when time-out occurs

Exit

```
If time-out was NOT scheduled then
    ESI = 0 (This is useful since 0 = NO TIME-OUT SCHEDULED)
else
    ESI = Time-out handle (used to cancel time-out)
```

Uses

ESI, Flags

Callback

EBX = Current VM handle (VM time-out was scheduled for)
ECX = Number of EXTRA milliseconds that have elapsed
EDX = Reference data
EBP -> Client register structure
Procedure may corrupt **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, and Flags.

Update_System_Clock

Description

This service must be called only by the Virtual Timer Device. If more than one device calls this service, then the VMM timing services will not behave correctly. The timer calls this procedure to update the current system time and the current VM's execution time. The value passed in **ECX** is the number of milliseconds that have elapsed since the last call to this service. In other words, if the current system time is *n*, then, after a call to **Update_System_Clock,** the current system time would be *n*+**ECX**.

This service assumes interrupts are disabled!

Entry

ECX = Elapsed time in milliseconds

Uses

Flags

Chapter 28 Processor Fault and Interrupt Services

The discussion of services providing general support for processor faults and interrupts are presented in the following order:

- **Get_Fault_Hook_Addrs**
- **Get_NMI_Handler_Addr**
- **Hook_NMI_Event**
- **Hook_V86_Page**
- **Set_NMI_Handler_Addr**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

Get_Fault_Hook_Addrs

Description Returns the address of the V86 mode, PM application, and VMM reenter fault handlers for a specified fault. If the fault does not have a handler, then this procedure will return 0. You cannot get the hook address for interrupt 2 (NMI). You must use the **Get/Set_NMI_Handler_Addr** services to hook interrupt 2.

Entry **EAX** = Interrupt number

Exit

```
If carry clear then
    EDX = Address of V86 Mode App fault handler (0 if none installed)
    ESI = Address of Prot Mode App fault handler (0 if none installed)
    EDI = Address of VMM Re-enter fault handler (0 if none installed)
else
    ERROR:  Invalid fault number
```

Uses Flags

Get_NMI_Handler_Addr

Description

If a VxD needs to hook the Non-Maskable Interrupt (NMI), it must first call this service to get the current NMI handler address, save the address so the current handler can be chained to it, and then set the new address.

Notice that your NMI interrupt handler can only touch local data in the device's VxD_LOCKED_DATA_SEG. It cannot touch memory in a VM handle, V86 memory, or any other memory. It also cannot call *any* services, *including* services that can be called during normal hardware interrupts. Because an NMI can occur at any time, it is difficult to do much of anything during interrupt time that is guaranteed not to reenter a non-reentrant procedure or affect a data structure.

Most NMI handlers will want to have an NMI event handler. This handler is similar to a normal event handler except that you only need to hook the NMI event chain once instead of scheduling an event every time. Every NMI event handler will be called every time an NMI occurs. Thus, most NMI interrupt routines simply detect that the NMI is for them and set a variable that their NMI event handler uses to perform some function. For example:

```
Initialization:
    VMMcall Get_NMI_Handler_Addr
    mov     [NMI_Chain_Addr], esi
    mov     esi, OFFSET32 My_NMI_Handler
    VMMcall Set_NMI_Handler_Addr
    mov     esi, OFFSET32 My_NMI_Event
    VMMcall Hook_NMI_Event
clc
ret

My_NMI_Handler:
    in    al, My_Stat_Port
    test  al, My_Int_Mask
    jz    SHORT MNH_Exit
    inc   [NMI_From_Me]
MNH_Chain:
    jmp   [NMI_Chain_Addr]

My_NMI_Event:
    xor   al, al
    xchg  al, [NMI_From_Me]
    test  al, al
    jz    SHORT NME_Exit
 (Do something here - NMI from my device)
MNE_Exit:
    ret
```

Entry

None

Exit

ESI = Offset of current NMI handler

Uses ESI, Flags

Hook_NMI_Event

Description See the documentation mentioned earlier in this chapter on **Get_NMI_Handler_Addr** for information on this service.

Entry **ESI** = Address of NMI event procedure

Exit None

Uses Flags

Callback **EBX** = Current VM handle
EBP -> Client register structure

Procedure may corrupt **EAX, EBX, ECX, EDX**

Hook_V86_Fault, Hook_PM_Fault, Hook_VMM_Fault

Description These services replace the fault handler procedure address with the procedure supplied. They will return the old fault handler's address or 0 to indicate that there was no previous fault handler. If the value returned in ESI is non-zero, then you may chain to the next handler with ALL REGISTERS PRESERVED. Your handler can "eat" a fault without chaining by executing a near return (not an iret) and can modify **EAX, EBX, ECX, EDX, ESI,** and **EDI.**

If you hook a fault during the **Sys_Critical_Init** phase of device initialization, your fault handler will be "behind" any VMM fault handler. If the VMM cannot properly handle a fault (for example, a General Protection fault), then it will chain to the next handler. By hooking GP faults during **Sys_Critical_Init** your VxD can intercept any GP fault that would otherwise crash the current VM. Any hooks installed after **Sys_Critical_Init** will be placed "in front of" the default VMM fault handlers. This allows devices to examine faults before they are processed by the VMM.

Note that the processor Non-Maskable Interrupt (NMI) must be hooked using the **Get/Set_NMI_Addr** services (do not call Hook_xxx_Fault with **EAX** = 2). Also, hardware interrupts should be hooked using the Virtual Programmable Interrupt Controller Device (VPICD). A VxD should NOT attempt to circumvent the VPICD using these services.

For version 3.0 of enhanced Windows, the largest interrupt number available is 4FH. Interrupts 00H-1FH are reserved by Intel for processor faults. Interrupts 20H-2FH are reserved by enhanced Windows. Interrupts 50H-5FH are used by the VPICD. Interrupts 40H and 41H are used by the debugger. Interrupts 42H-4FH are free for use by VxDs.

Entry

EAX = Interrupt number
ESI = Procedure offset

Exit

```
If carry clear then
        ESI = Old procedure offset (0 if none)
else
        ERROR:  Invalid fault number in EAX
```

Uses

ESI, Flags

Callback

Interrupts disabled
EBX = Current VM handle
If fault from V86 or PM app then
    EBP -> Client_Register_Strucuture
else
    VMM reentered — Only asynchronous services may be called.
    EBP -> VMM re-entrant fault stack frame

If your handler chains, then it must preserve *all* registers (even registers *not* documented as entry conditions to this callback).

Hook_V86_Page

Description

This service allows VxDs to intercept page faults in portions of the V86 address space of every virtual machine. It is used by devices such as the Virtual Display Device to detect when particular address ranges are accessed.

You must specify a page number and address of a callback routine to this service. If it is installed successfully, your hook will be called every time a page fault occurs in *any* VM on that page. See the memory manager **_Modify_Pages** documentation in Chapter 19, "Memory Management Services," for making hooked pages not present and for registering the ownership of pages.

The callback routine is responsible for mapping memory at the location of the page fault or crashing the VM. In unusual circumstances, it may be appropriate to map a NULL page at the faulting address page. See the memory manager documentation for details on mapping memory and mapping NULL pages.

NOTE Do not rely on the contents of the **CR2** (page fault) register. Use the value passed to your call-back in **EAX**.

Entry

EAX = Page number (A0h - FFh)
ESI = Address of trap routine

Exit

```
If carry flag set then
    ERROR:  Invalid page number or page already hooked
 else
    Page hooked
```

Uses

Flags

Callback

EAX = Faulting page number
EBX = Current VM handle
EBP does NOT point to the client register structure.

Procedure may corrupt **EAX, EBX, ECX, EDX, ESI, EDI**, and Flags

Set_NMI_Handler_Addr

Description

See the documentation mentioned earlier in this chapter on **Get_NMI_Handler_Addr** for information on this service.

Entry

ESI = Offset of new NMI handler

Exit

None

Uses

Flags

Chapter 29 Information Services

These services return the requested information without instigating any other action. They provide information on the following:

- VM handles
- The VMM reenter count
- HMA XMS
- Installation status of the debugger

They are described here in the following order:

- **Get_Cur_VM_Handle**
- **Get_Next_VM_Handle**
- **Get_Sys_VM_Handle**
- **Get_VMM_Reenter_Count**
- **Get_VMM_Version**
- **GetSet_HMA_Info**
- **Test_Cur_VM_Handle**
- **Test_Debug_Installed**
- **Test_Sys_VM_Handle**
- **Validate_VM_Handle**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

Get_Cur_VM_Handle

Description This service returns the handle to the currently running VM. It is valid to call this service at interrupt time.

Entry None

Exit **EBX** = Current VM handle

Uses **EBX**, Flags

Get_Next_VM_Handle

Description VMM maintains a list of all valid VM handles. This service provides a means of scanning the list easily. Normally, code that uses this service looks something like this:

```
VMMcall Get_Cur_VM_Handle
 Scan_Loop:
...
(Do something to VM state)
...
VMMcall Get_Next_VM_Handle
VMMcall Test_Cur_VM_Handle
jne     Scan_Loop
```

This allows the state of every VM to be modified. However, there are also other uses for this service. There is no guaranteed ordering of the list other than the fact that each VM will appear in the list only once. Notice also that the list is circular so you will need to test for the end case (Next VM = First VM). It is valid to call this service at interrupt time.

Entry **EBX** = VM handle

Exit **EBX** = Next VM handle in VM list

Uses **EBX**, Flags

Get_Sys_VM_Handle

Description This service returns the System VM handle. It is valid to call this service at interrupt time.

Entry None

Exit **EBX** = System VM handle

Uses **EBX**, Flags

Get_VMM_Reenter_Count

Description This service is used to determine if the VMM has been reentered from an interrupt. The normal situation for reentering VMM is from a hardware interrupt, page fault, or other processor exception. Since most VMM services are non-reentrant, this test should be used to determine if other VMM services can be called or if a global event should be scheduled. Notice that the **Call_Global_Event** service tests this condition automatically and will schedule an event if VMM has been reentered.

Entry None

Exit **ECX** = 0 indicates VMM has NOT been re-entered. If != 0 then
ECX = # of times re-entered

Uses Flags

Get_VMM_Version

Description This service returns the Windows/386 VMM version.

Entry None

Exit **AH** = Major version number (3)
AL = Minor version number (0)
Carry flag clear

Uses **EAX** , Flags

GetSet_HMA_Info

Description This service returns and sets information related to the HMA XMS region.

This service is intended to assist the XMS driver that is part of the V86MMGR device. It allows the protected-mode XMS code to find out if there was a global HMA user in before Windows/386 was started and allows access to the Enable count variable (Get and Set). This service is always valid (i.e., not restricted to initialization).

Entry

ECX == 0 Get
ECX != 0 Set
DX = A20 enable count to set for Win386 loader
NOTE THAT THE GLOBAL HMA FLAG CANNOT BE SET. It is not appropriate or valid to set this.

Exit

If Get
EAX == 0 if WIN386 DID NOT allocate the HMA (GLOBAL HMA User)
EAX != 0 if WIN386 allocated the HMA (NO GLOBAL HMA User)
EDX = A20 enable count before Win386 came in
If Set

Uses

EAX, **EDX**, Flags

Test_Cur_VM_Handle

Description

This routine tests to see if the given VM handle is the handle of the currently running VM. It is valid to call this service at interrupt time.

Entry

EBX = VM handle to test

Exit

Zero flag is set if VM handle passed in
is currently running VM's handle. (je **Is_Cur_VM**)

Uses

Flags

Test_Debug_Installed

Description

Tests internal flag that indicates whether a debugger exists or not. It is valid to call this service at interrupt time.

Entry

None

Exit

Zero flag = Debugger NOT installed (i.e., jz No_Debug_Installed)

Uses

Flags

Test_Sys_VM_Handle

Description This routine tests to see if the given VM handle is the handle of the system VM. It is valid to call this service at interrupt time.

Entry **EBX** = VM handle to test

Exit Zero flag is set if VM handle passed in is system VM's handle. (je **Is_Sys_VM**)

Uses Flags

Validate_VM_Handle

Description This service is used to test the validity of a VM handle. This service can be called at interrupt time.

Entry **EBX** = VM handle to test

Exit If carry flag set then
    ERROR:VM handle is invalid
else
    Value in **EBX** is a valid VM handle

Uses Flags

Chapter 30

Initialization Information Services

These services provide access to the SYSTEM.INI file and the environment variables. Configurable VxDs will use these services to get their configuration parameters. They are described here in the following order:

- **Convert_Boolean_String**
- **Convert_Decimal_String**
- **Convert_Fixed_Point_String**
- **Convert_Hex_String**
- **Get_Config_Directory**
- **Get_Environment_String**
- **Get_Exec_Path**
- **Get_Machine_Info**
- **Get_Next_Profile_String**
- **Get_Profile_Boolean**
- **Get_Profile_Decimal_Int**
- **Get_Profile_Fixed_Point**
- **Get_Profile_Hex_Int**
- **Get_Profile_String**
- **Get_PSP_Segment**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

Convert_Boolean_String (Initialization only)

Description

This service attempts to determine if the string pointed to by **EDX** is TRUE or FALSE. There are many valid values for TRUE and FALSE. A short list of valid values for TRUE are:

True, Yes, On, 1

For false they include:

False, No, Off, 0

This list may grow to include other words such as "oui" and "ja."
This service is only valid during initialization.

Entry

EDX = Pointer to ASCIIZ string to convert to boolean

Exit

```
If carry clear then
    EAX = 0 if FALSE, -1 if TRUE, zero flag NOT set
else
    String was not a valid boolean (EAX not changed)
```

Uses

Flags, **EAX**

Convert_Decimal_String (Initialization only)

Description

This service converts a string that contains a decimal value and returns the value in **EAX**. It also returns a pointer to the character that terminated the decimal integer value. This is useful for parsing entries such as:

FOO=100,300

since the 100 would be returned with **EDX** pointing to the ",". The pointer could be incremented one byte and, then, this service called again to evaluate the second number.

Notice that a NULL string or a string that does not contain a valid decimal integer will return 0 and **EDX** will not be advanced since the first character of the string terminated the analysis. This service is only valid during initialization.

Entry

EDX = Pointer to ASCIIZ string to convert to integer

Exit

EAX = Value of decimal string
EDX = Pointer to terminating character (non-valid decimal char)

Uses

EAX, **EDX**, Flags

Convert_Fixed_Point_String (Initialization only)

Description

This service returns the value of a fixed point decimal number string pointed to by **EDX.** Use **Get_Profile_String** to initialize **EDX** to point to the string to be parsed. Fixed Point is zero or more decimal digits followed by a terminator or a decimal point followed by zero or more decimal digits. The value returned is **ECX***10*<value of string>. Note that decimal digits beyond the accuracy specified by **ECX** are ignored in the value returned in **EAX,** but **EDX** points to the byte following the last valid ASCII decimal digit. Values that begin with a minus will evaluate to negative numbers. Positive values may optionally begin with a plus sign.

This service is only valid during initialization.

Entry

ECX = Number of decimal places
EDX = Pointer to ASCIIZ string to convert to integer

Exit

EAX = Value of fixed point string
EDX = Pointer to terminating character (non-valid character)

Uses

EAX, EDX, Flags

Convert_Hex_String (Initialization only)

Description

This service converts the string pointed to by **EDX** to Hexadecimal. Hexadecimal is zero or more hexadecimal digits (0-9, A-F) followed by a terminating character or a small or capital letter "h". The "h" has no effect on the value. **EDX** is left pointing to the next byte after the "h" or, if the "h" is not present, after the last valid hexadecimal digit. Use **Get_Profile_String** to set up **EDX** to point to the string to be parsed. This service is only valid during initialization.

Entry

EDX -> ASCIIZ string to convert to integer

Exit

EAX = Value of hexadecimal string
EDX advanced to terminating character (non-valid hex char)

Uses

Flags

Get_Config_Directory (Initialization only)

Description

This service returns a pointer to the directory that contains the configuration files for the enhanced Windows environment (such as SYSTEM.INI). The string returned is guaranteed to be a valid, fully qualified pathname that ends with a terminating "\" followed by a NULL (0) byte.

Entry None

Exit **EDX** = Pointer to ASCIIZ directory name

Uses **EDX**, Flags

Get_Environment_String (Initialization only)

Description This service takes a pointer to an ASCIIZ string that is the name of an environment variable and returns a pointer to an ASCIIZ string that is the value of that environment variable. Environment variables are set using the DOS **SET** command and should be of the format "**SET** *<variable name>=<variable value>*" with no intervening spaces between the variable name, the equal sign, and the variable value. Environment strings are an alternative way of setting parameters for virtual device drivers. In general, these should be used sparingly, as the environment is of limited size. Use environment strings only when the value is a global entity, used by more than one program or device driver. This service is only valid during initialization.

Entry **ESI** = pointer to ASCIIZ string environment variable name

Exit

```
If carry is set then
    Environment string was not found
else
    EDX = pointer to ASCIIZ string value of environment variable
```

Uses **EDX**, Flags

Get_Exec_Path (Initialization only)

Description This service returns a pointer to an ASCIIZ string that gives the full path by which WIN386.EXE was executed. It is used to locate files associated with the enhanced Windows environment or the virtual device drivers that are not in subdirectories indicated by the PATH environment variable. This service is only valid during initialization.

Entry None

Exit **EDX** = Pointer to ASCIIZ string of full path name + program name (program name is "WIN386.EXE")
ECX = Number of characters in string up to and including the last "\"

Get_Machine_Info (Initialization only)

Description This service returns information about the computer system running enhanced Windows.

Entry None

Exit
AH = DOS Major Version
AL = DOS Minor Version
BH = DOS OEM serial number
BL = Machine Model Byte (at F000:FFFE in system ROM)
HIGH 16 bits of **EBX** are other flags
GMIF_80486 EQU 10000h 80486 processor
GMIF_PCXT EQU 20000h PCXT accelerator
GMIF_MCA EQU 40000h Micro Channel
GMIF_EISA EQU 80000h EISA
EDX = Equipment flags (as returned from Int 11h)
ECX = 0 if not PS/2 or extended BIOS, else **ECX** contains a ring 0 linear address to System Configuration Parameters returned from BIOS service Int 15h, AH=C0h. See the PS/2 BIOS documentation for details on this structure.

Uses **EAX, EBX, ECX, EDX,** Flags

Get_Next_Profile_String (Initialization only)

Description This service, given a pointer to a profile string, will return a pointer to the next profile string with the key name provided. It is used by devices that have multiple entries with the same key name. First, use **Get_Profile_String** to get the first entry with a given key name and, then, use this service to get subsequent entries. Do not modify the string returned. This service is only valid during initialization.

Entry
EDX = Pointer returned from previous **Get_(Next)_Profile_String**
EDI = Pointer to key name string

Exit
```
If carry clear then
    EDX = NEXT string from SYSTEM.INI
else
    No more matching entries found
```

Uses EDX, Flags

Get_Profile_Boolean (Initialization only)

Description This service returns the value of a Boolean profile entry from the SYSTEM.INI file in **EAX**. If the profile string is not found, then **EAX** will not be modified. Profile entries are of the form:

```
[SectionName]
KeyName=<value>
```

That is, Section Name is delineated by square brackets and KeyName is followed by an equal sign. Neither name should have any spaces or nonprintable characters. The value following the equal sign can be in a number of formats. Boolean is "Yes," "No," "Y," "N," "True," "False," "On," "Off," "1," or "0"(foreign versions of Windows may add other language equivalents to the above). Logical TRUE returns -1 and logical FALSE returns 0.

This service is only valid during initialization.

Entry **EAX** = Default value
ESI = Pointer to section name string or 0 for [386enh]
EDI = Pointer to key name string

Exit

```
If carry set
    Entry not found or invalid boolean value
    EAX = Default value
else
      If value string was null,
           zero flag is set and
          EAX = Default value
     else
          EAX = 0 if FALSE, -1 if TRUE SYSTEM.INI entry value
```

Uses Flags

Get_Profile_Decimal_Int (Initialization only)

Description This service returns the value of a decimal profile entry from the SYSTEM.INI file in **EAX**. If the profile string is not found, then **EAX** will not be modified. Profile entries are of the form:

```
[SectionName]
KeyName=<value>
```

That is, SectionName is delineated by square brackets and KeyName is followed by an equal sign. Neither name should have any spaces or non-printable characters. The value following the equal sign must be a decimal value. It can begin optionally with a plus (+) or minus (-) and must contain all decimal digits with no embedded spaces or decimal points.

This service is only valid during initialization.

Entry

EAX = Default value (optional)
ESI = Pointer to section name string or 0 for [386enh]
EDI = Pointer to key name string

Exit

```
If carry is set
    Entry was NOT found
    EAX = Default value (value passed to this procedure)
else
    If value string was null, zero flag is set and
         EAX = Default value
    else
        EAX = Value of SYSTEM.INI entry
```

Uses

Flags

Get_Profile_Fixed_Point (Initialization only)

Description

This service returns the value of a fixed point decimal number profile entry from the SYSTEM.INI file in **EAX**. If the profile string is not found, then **EAX** will not be modified. Profile entries are of the form:

[SectionName]
KeyName=<value>

That is, SectionName is delineated by square brackets and KeyName is followed by an equal sign. Neither name should have any spaces or nonprintable characters. The value following the equal sign can be in a number of formats. Fixed Point values may begin with an optional plus (+) or minus (-) followed by zero or more decimal digits followed by a terminating character or by a decimal point followed by zero or more decimal digits. The value returned is 10^**ECX***<value of string>.

This service is only valid during initialization.

Entry

EAX = Default value
ECX = Number of decimal places
ESI = Pointer to section name string or 0 for [386enh]
EDI = Pointer to key name string

Exit

```
If carry is set
    Entry was NOT found
    EAX = Default value (value passed to this procedure)
```

```
else
    If value string was null, zero flag is set and
    EAX = Default value
else
    EAX = Value of SYSTEM.INI entry
```

Get_Profile_Hex_Int (Initialization only)

Description

This service returns the value of a hexadecimal number profile entry from the SYSTEM.INI file in **EAX**. If the profile string is not found, then **EAX** will not be modified. Profile entries are of the form:

[SectionName]
KeyName=<value>

That is, SectionName is delineated by square brackets and KeyName is followed by an equal sign. Neither name should have any spaces or nonprintable characters. The value following the equal sign can be in a number of formats. Hexadecimal is zero or more hexadecimal digits (0-9, A-F) followed by a terminating character or a small or capital letter "h." The "h" has no effect on the value. If the value following the equal sign is not a valid hexadecimal number, **EAX** is unchanged.

This service is only valid during initialization.

Entry

EAX = Default value (optional)
ESI = Pointer to section name string or 0 for [386enh]
EDI = Pointer to key name string

Exit

```
If carry is set
    Entry was NOT found
    EAX = Default value (value passed to this procedure)
else
     If value string was null
          zero flag is set
          EAX = Default value
    else
          EAX = Value of SYSTEM.INI entry
```

Uses

Flags

Get_Profile_String (Initialization only)

Description

This service searches the initialization file for a specified entry and returns a pointer to a string. Do *not* modify the string in place. The pointer returned points into the initialization file data area. If you need to modify the string, you must first copy it and, then, modify it. This service is only valid during initialization.

Entry

EDX = Pointer to default string (optional)
ESI = Pointer to program name string or 0 for [386enh]
EDI = Pointer to key name string

Exit

```
If carry clear
    EDX = Pointer to ASCIIZ string from SYSTEM.INI
else
    EDX is unchanged
```

Uses

Flags, may change **EDX**

Get_PSP_Segment (Initialization only)

Description

This service returns the segment of the WIN386.EXE PSP. Use it to locate PSP values other than the EXEC path and environment variables since separate services are available for retrieving those ASCIIZ strings. Notice that a segment value is returned. To convert the segment to an address, shift the value left by 4 bits. This service is only valid during initialization.

Entry

None

Exit

EAX = Segment of WIN386.EXE PSP (high word always = 0)

Uses

EAX, Flags

Chapter 31 Linked List Services

These services provide a convenient set of routines for managing a linked-list data structure. They are described here in the following order:

- **List_Allocate**
- **List_Attach**
- **List_Attach_Tail**
- **List_Create**
- **List_Deallocate**
- **List_Destroy**
- **List_Get_First**
- **List_Get_Next**
- **List_Insert**
- **List_Remove**
- **List_Remove_First**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

List_Allocate

Description This service allocates a new node for the list specified by **ESI**. The contents of the node are undefined (probably nonzero). Normally, a node is immediately attached to the list through the **List_Attach** or **List_Insert** services after it has been allocated.

Entry **ESI** = List handle

Exit

```
If list was created with LF_Alloc_Error flag then
    If carry clear then
           EAX -> New node
    else
           Error:Could not allocate node
```

```
else
    EAX -> New node
    (Current VM crashed if node can not be allocated - Service
    never returns to caller)
```

Uses **EAX**, Flags

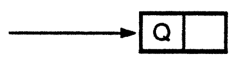
List_Attach

Description This service attaches a list node to the head (i.e., front) of a list. Notice that **EAX** must point to a node that was allocated using **List_Allocate**.

Nodes can be attached to any list that has the same size node. This can be used, for example, to move a node from one list to another.

Example Assume we have the following list:

List_Attach with EAX



Produces the following list:

Figure 31.1 SERV_05.EPS

Entry **ESI** = List handle
EAX -> Node

Exit Node attached to list

Uses Flags

List_Attach_Tail

Description This service attaches a list node to the tail (i.e., end) of a list. **EAX** must point to a node that was allocated using **List_Allocate**.

Nodes can be attached to any list that has the same size node. This can be used, for example, to move a node from one list to another.

Example Assume we have the following list:

List_Attach_Tail with EAX

Produces the following list:

Figure 31.2 SERV_11.EPS

Entry

ESI = List handle
EAX -> Node to insert

Exit

Node inserted at tail (end) of list

Uses

Flags

List_Create

Description

This service is used to create a new list structure. This service returns a list handle that is used when calling all subsequent list services.

Lists normally allocate nodes from a "pool" of free nodes. This prevents the overhead that would be incurred by calling **_HeapAlloc** and **_HeapFree** for every list allocation and deallocation. Once a node is created, it is never destroyed. Instead, **List_Deallocate** places the node back in the free pool. The node can then be reclaimed quickly when **List_Allocate** is called.

If the size of the list nodes are large, you should force them to be allocated from the system heap by setting the **LF_Use_Heap** flag. All allocate/deallocate calls for lists created in this way will use **_HeapAlloc** and **_HeapFree** to create and destroy nodes.

If you want to be able to access a list during hardware interrupts, you should set the **LF_Async** flag. This forces list operations to be atomic operations (they cannot be re-entered). If you select this option, you must call list services with *INTERRUPTS DISABLED* or an error will occur. You must disable interrupts even if you are not calling the list service from an interrupt. Remember, always use **pushf/CLI/popf** to disable interrupts. Never explicitly use STI unless other documentation states that this is permissable. Notice that since **_HeapAllocate** and **_HeapFree** cannot be called from a hardware interrupt, you cannot select this option and **LF_Use_Heap.**

The **LF_Alloc_Error** flag should be used if you would like to recover from an allocation error (i.e., out of memory). The default behavior for a failed allocation is to crash the current VM. However, if your VxD would like to have the allocation return an error, set this flag. If this option is selected, then **List_Allocate** will return with the Carry flag set when an allocation fails. Otherwise, it will crash the current virtual machine whenever it cannot allocate a new node.

Entry

EAX = Flags
LF_Use_Heap - All data on system heap (Can't use with LF_Async)
LF_Async - List services can be called at interrupt time
LF_Alloc_Error - Return from alloc with carry set if can't allocate
ECX = Node size

Exit

```
If Carry Flag is clear then
    ESI = List handle
else
    Error: Unable to create list
```

Uses

ESI, Flags

List_Deallocate

Description

This service places a list node in the free memory pool. Once a node has been deallocated, it should not be referenced again. You must remove the node from any list to which it is attached before deallocating it.

Entry

ESI = List handle
EAX -> List node

Exit

EAX is undefined

Uses

EAX, Flags

List_Destroy

Description

This service deallocates all nodes on a list and destroys the list handle. Once a list has been destroyed, its handle is no longer valid.

Entry

ESI = List handle

Exit

ESI is undefined
List is destroyed, all nodes deallocated.

Uses ESI, Flags

List_Get_First

Description This service returns a pointer to the first node in a list. If the list is empty, it will return 0 and the Zero Flag will be set.

Entry ESI = List handle

Exit

```
If ZF is clear then
    EAX -> First node in list
else
    List is empty. EAX = 0.
```

Uses EAX, Flags

List_Get_Next

Description This service returns the next node in a list. It is used to traverse the list when searching for a specific element. If the end of the list is reached, it will return 0 and the Zero Flag will be set.

Typically, this service is used in conjunction with List_Get_First to scan an entire list.

```
EXAMPLE:
  BeginProc Scan_My_List
        mov      esi, [My_List_Handle]
        VMMcall List_Get_First
        jz       SHORT Scan_Done
  Scan_Loop:
  (Do something with EAX here)
        VMMcall List_Get_Next
        jnz      Scan_Loop
  Scan_Done:
        ret
  EndProc Scan_My_List
```

Entry ESI = List handle
EAX -> Node

Exit If ZF is clear then
EAX -> Next node in list
else
End of list reached. EAX = 0.

Uses EAX, Flags

List_Insert

Description This service inserts a node at a specified point in a list. The caller must specify two nodes: the node to be inserted in **EAX**, and a position to insert the node *after* in **ECX**. This means that node **EAX** will occupy the position in the list immediately after node **ECX**. If **ECX** is zero, then node **EAX** will be inserted at the head of the list.

Nodes can be inserted in any list that has the same size node. This can be used, for example, to move a node from one list to another.

Example Assume we have the following list:

List_Insert with **ECX** pointing to Y-node and **EAX** pointing to Q-node produces the following list:

Figure 31.3 SERV_08.EPS

Entry
ESI = List handle
EAX -> Node to insert
ECX -> Node to insert after (0 to attach to head)

Exit Node inserted in list

Uses Flags

List_Remove

Description This service removes a specified node from a list.The node will *not* be deallocated by this service. It is up to the caller to deallocate the node or attach it to another list (it can only be attached to a list with node size equal to the original list).

Example Assume we have the following list:

List_Remove with **EAX** pointing to Y-node produces the following list:

Figure 31.4 SERV_14.EPS

Entry **ESI** = List handle
EAX -> Node to remove from list

Exit Node removed from list

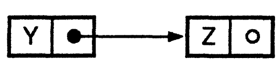
Uses Flags

List_Remove_First

Description This service removes the first node from a list. Notice that the node is *not* deallocated by this service. It is up to the caller to deallocate the node or attach it to another list (it can only be attached to a list with node size equal to the original list).

Example Assume we have the following list:

List_Remove_First produces the following list:



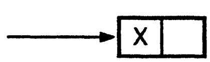
and EAX 

Figure 31.5 SERV_16.EPS

Entry **ESI** = List handle

Exit

```
If Zero Flag is clear then
    EAX -> Node that has been removed from list
else
    List is empty and EAX = 0
```

Uses EAX, Flags

Chapter 32 Error Condition Services

These error services are used by VxDs when they have detected the VM to be in an unrecoverable state. Examples of situations that might lead to such a state include an attempted VM execution of a protected instruction or an operation which might fail due to lack of memory. The services are described here in the following order:

- **Crash_Cur_VM**
- **Fatal_Error_Handler**
- **Fatal_Memory_Error**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

Crash_Cur_VM

Description

This service will crash the current VM. It is to be called when a catastrophic error has occured in the VM, such as executing an illegal instruction or attempting to program a piece of hardware in a way incompatible with the device virtualization.

If the system VM is the current VM, enhanced Windows will exit with a fatal error without explicitly crashing the other VMs.

Entry

None

Exit

None

Fatal_Error_Handler

Description

This service is called (or jumped to) when a fatal error is detected. It returns to real mode and, optionally, prints out an error message. You can hang the computer by selecting the **EF_Hang_On_Exit** flag (defined in VMM.INC).

All the devices are informed about the exit before returning to real mode.

The **Fatal_Error** macro supplied in VMM.INC is a convenient way of calling this service.

Examples:

```
Fatal_Error                              ; This exits with no error message
Fatal_Error <OFFSET32 My_Err_Msg>        ; Exits and prints error message
```

Entry **ESI** = Ptr ASCIIZ string to display (0 if none)
EAX = Exit flags to send to the loader (real mode exit code)
Bit 0 = 1 - Hang system on exit to real mode
Others undefined and must be 0

Exit None

Uses All registers

Fatal_Memory_Error

Description This routine calls the **Fatal_Error_Handler** with exit flags equal to zero and the message "Not Enough Memory to Run Windows/386". It should be called during **Device_Init**, **Init_Complete**, or **Sys_VM_Init** if there is not enough memory to initialize.

Entry None

Exit None

Uses All registers

Chapter 33 Miscellaneous Services

The services discussed in this chapter provide functions not easily categorized such as hooking another VxDs API and sending system control messages. They are provided here in the following order:

- **Begin_Reentrant_Execution**
- **End_Reentrant_Execution**
- **Hook_Device_Service**
- **Hook_Device_V86_API**
- **Hook_PM_Device_API**
- **Map_Flat**
- **MMGR_SetNULPageAddr**
- **Simulate_Pop**
- **Simulate_Push**
- **System_Control**

See Chapter 16. "Overview of Windows in 386 enhanced mode" and Chapter 17, "Virtual Device Programming Topics" for general environment discussions.

Begin_Reentrant_Execution

Description

THIS IS A VERY DANGEROUS SERVICE. BE VERY CAREFUL WHEN CALLING IT. Most virtual devices have no reason to use this service. Do NOT use this service to avoid scheduling events on hardware interrupts.

It is intended to be used by devices that hook VMM Faults (re-entrant processor exeptions) that must call non-asynchronous VMM or VxD services or execute a VM. This would be valid to use, for example, if a VxD provided a ring 0 software interrupt interface (although this is NOT RECOMMENDED — You should provide device services through the Win386 dynamic-linking mechanism). It would be INVALID to use this service during a hardware interrupt (such as a timer or disk interrupt).

Entry None

Exit ECX = Old reentrancy count (must be passed to **End_Reentrant_Execution**)

Uses ECX, Flags

End_Reentrant_Execution

Description A VxD that calls **Begin_Reentrant_Execution** must call this service before returning.

Entry ECX = Reentrancy count returned from **Begin_Reentrant_Execution**

Exit None

Uses Flags

Hook_Device_Service

Description This service allows one device to monitor or replace a device service. *extreme care* must be taken here not to destroy the functionality of the device whose routine is being monitored or replaced. This service also allows VMM services to be hooked (the VMM is device 1).

Hooking a service is often useful for monitoring the activities of other devices. For example, if a device needed to know whenever a VM was set into background mode, it could use the following code:

```
(Initialization code)
            mov         eax, Set_Time_Slice_Priority
            mov         esi, OFFSET32 My_Hook_Proc
            VMMcall Hook_Device_Service
            jc     Error!
            mov       [Real_Proc], esi

BeginProc My_Hook_Proc
            test          eax, VMStat_Background
            jz      SHORT MHP_Chain
            pushad
            (Do something here)
            popad
MHP_Chain:
            jmp     [Real_Proc]
EndProc My_Hook_Proc
```

Every time a VxD calls **Set_Time_Slice_Priority**, the **My_Hook_Proc** procedure will be called. The hook procedure should normally chain to the actual device or VMM service although this is not required. Also, be sure to save and restore any registers in your hook procedure.

You will notice that the sample initialization code moves **Set_Time_Slice_Priority** into **EAX**. Remember, services are defined as EQUATES, not external procedure references. Thus, **Set_Time_Slice_Priority** is just a number. (VMM device ID< < 16 + Service number).

Your hook must preserve all registers that are not modified by the service you have hooked. Also, if flags are passed as an entry or exit parameter, your hook procedure must also preserve the flags.

Be careful about hooking C calling convention (stack-based) services. If you want to examine the "back end" of a C calling convention service, you will need to copy the entire parameter stack frame before calling the actual service.

More than one VxD can hook a device service. The last hook installed will be the first one called.

Entry

EAX = Device ID << 16 + Service number (use service equate)
ESI = New procedure

Exit

```
If carry clear then
    ESI = Old dynalink procedure
else
    ERROR.  Invalid Device or Service number
```

Uses

ESI, Flags

Hook_Device_V86_API, Hook_PM_Device_API

Description

These services allow a VxD to hook another virtual device's V86 or protected mode API interface.You are responsible for chaining to the real API handler. Be careful to preserve the **EBX** and **EBP** registers when calling the next handler in the chain.

Most VxDs will never need to hook another virtual device's API procedure. These services are provided mainly as a mechanism for devices that may be developed in the future to intercept API calls to other virtual devices. For example, a new version of the Virtual Mouse Device may need to intercept calls to the Virtual Display Device so that it can save and restore the mouse cursor. In such a case, these services could be used.

Entry

EAX = Device ID
ESI = Offset of new API handler

Exit

```
If carry clear then
    ESI = Offset of previous API handler (used to chain to next
          handler)
else
    ERROR: Device does not support API interface
```

Uses

ESI, Flags

Callback

EBX = Current VM handle
EBP -> Client register structure
(Same parameters as standard API entry point)

Map_Flat

NOTE Please be advised that the following description has been indentified as out of date in some repects though updated information was unavailable at the time of this printing.

Description

This service provides a convenient way of converting a SEGMENT:OFFSET or SELECTOR:OFFSET pair into a linear address. **Map_Flat** works only for the current VM. It determines whether the value passed to it is a V86 segment or a PM selector by the execution mode of the current VM. This allows VxDs to use identical code for PM and V86 handlers. For example, assume a VxD wanted to simulate DOS reads in both V86 and protected mode. It would hook both the V86 and PM **int** chains with the same procedure:

```
VxD_DOS_Read_Hook:
        cmp     [ebp.Client_AH], 3Fh    ; Q: Is it a read
        jne     SHORT VxD_DRH_Reflect   ;    N: Reflect it
                                        ;    Y: DS:DX -> Read buffer
        mov     ax, (Client_DS SHL 8) + Client_DX
        VMMcall Map_Flat        ; EAX = Lin addr of DS:DX
        . . .
        (Do something useful here)
        . . .
        clc                             ; Eat this int 21h
        ret VxD_DRH_Reflect:
        stc
        ret
```

Notice that the above procedure does *not* need to examine the VM's execution state. By calling **Map_Flat**, it converts the **DS:DX** pointer into a valid linear address regardless of the VM's execution mode.

The **Client_Ptr_Flat** macro will generate this code automatically. For the proceding example, you would use:

```
Client_Ptr_Flat eax, DS, DX
```

The first parameter specifies the 32-bit register to contain the linear address. The second parameter specifies the client's segment. The third parameter is optional and specifies the offset register (if blank, then an offset of 0 is assumed).

Entry EAX = ~~~~~~~~~~~~~~~~

Exit EAX = Ring 0 linear address

Uses Flags, EAX

MMGR_SetNULPageAddr

DESCRIPTION This call is used to set the physical address of the system nul page.

It can be called at device INIT time to set the address of a KNOWN non-existant page in the system. This is usually called by the V86MMGR device because he does memory scans and therefore has a good idea about what a good page will be.

ENTRY EAX is PHYSICAL address for NUL Page (Page number << 12)

EXIT None

USES Flags

Simulate_Pop

Description Returns the WORD or DWORD at the top of the current VM's client stack and adds 2 or 4 to the client's **SP**.

Entry None

Exit EAX = Word popped from application's stack (high word 0 if use 16 app)

Uses EAX, Client_ESP, Flags

Simulate_Push

Description Pushes a WORD or DWORD onto the current VM's client stack and decrements the VM's **SP** by 2 or 4.

Entry

```
If in V86 mode or 16 bit PM application then
    AX = WORD to push
else
    EAX = DWORD to push
```

Exit (D)WORD pushed on application program's stack

Uses **Client_ESP**, Flags

System_Control

Description This service sends system control messages to all the VxD's and for some messages, to parts of VMM as well. Notice that incorrect usage of the system control messages can cause erratic behavior by the system. For example, only the Shell device should initiate **Create_VM** and **Destroy_VM** messages. Also notice that when a **Set_Device_Focus** message is done with a device ID of zero, all devices with a settable focus must set their focus to the VM indicated.

The valid **System_Control** messages are as follows:

Initialization	**Sys_Critical_Init** **Device_Init** **Init_Complete**
System VM creation	**Sys_VM_Init** **Sys_VM_Terminate**
System VM destruction (WIN386 exit)	**System_Exit** **Sys_Critical_Exit**
Other VM creation	**Create_VM** **VM_Critical_Init** **VM_Init**
Other VM destruction	**VM_Terminate** **VM_Not_Executable** **Destroy_VM**
VM state changes	**VM_Suspend** **VM_Resume** **Set_Device_Focus**
Special messages	**Reboot_Processor** **Debug_Query**

The control calls that are valid for devices to issue areas follows:

```
Create_VM                (used by SHELL)
Destroy_VM               (used by SHELL)
Set_Device_Focus
```

Entry

EAX = System control message
EBX = VM handle (if needed by message)
ESI,EDI,EDX = message specific parameter, such as Device ID (for Set_Device_Focus message)
ECX register is used by this service and cannot contain any parameter that will be passed through to the devices.

Exit

```
Carry Set
    Call failed
Carry Clear
    Call Succeeded
    If Entry EAX = Create_VM
           EBX = New VM handle created
```

Uses

Flags, EBX if Create_VM

Chapter 34 Shell Services

The Shell services provide a way for VxDs to communicate with the user. This chapter presents descriptions of the Shell services in the following order:

- **SHELL_Event**
- **SHELL_Get_Version**
- **SHELL_Message**
- **SHELL_Resolve_Contention**
- **SHELL_SYSMODAL_Message**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

SHELL_Event

Description

This procedure posts an event in the windows shell to VMDOSAPP. This service is primarily for SHELL to WINOLDAPP COMMUNICATION. The VDD also sends a couple messages to WINOLDAPP other devices should have no use for this service.

Entry

EBX is VM Handle for Event
ECX is event #
AX = wParam for event
High 16 bits **EAX** special boost flags
ESI is callback procedure for event (==0 if none)
EDX is reference data for event callback

Exit

Carry Clear
  Event placed in queue
  EAX is "Event Handle" of event ONLY VALID IF ENTRY **ESI** != 0
Carry Set
  Event not placed
VMDOSAPP not present
Insufficient memory for placement

Callback

Carry Set
Event could not be placed in VMDOSAPP queue
EDX = reference data
NOTE THAT **EBX** != VM Handle of event!
Carry Clear
Called when VMDOSAPP signals event processing complete
EBP -> VMDOSAPP Client frame so registers can be accessed
EDX = reference data
NOTE THAT **EBX** != VM Handle of event!

Uses

Flags, **EAX**

SHELL_Get_Version

Description

This procedure returns the version of the Shell VxD.

Entry

None

Exit

AH = Major version
AL = Minor version
Carry Flag set

Uses

EAX, Flags

SHELL_Message

Description

This procedure is called to put up messages. Refer to SHELL.INC and the *Microsoft Windows Software Development Kit* for information on message box parameters.

Entry

EBX = VM Handle of VM responsible for message
EAX = Message box flags (SEE MB_xxxx in SHELL.INC)
ECX -> NUL terminated Message Text
EDI -> NUL terminated caption Text == 0 for standard caption
-> NUL for No caption
ESI -> Callback procedure to call with response when dialog is finished == 0 if no call back desired
EDX = Reference data for callback

Exit

Carry Clear
EAX is "Event Handle" of message
Carry Set
Message cannot be displayed (insufficient memory)

CALLER MAY WISH TO CALL **SHELL_SYSMODAL_MESSAGE** IN THIS CASE. **SHELL_Sysmodal_Message** will not fail.

Callback

Called when message box is complete
EAX = Response code from dialog box (SEE IDxx in SHELL.INC)
EDX = reference data

Uses

Flags, **EAX**

SHELL_Resolve_Contention

Description

This procedure is called to resolve contention. It displays a dialog box in which the user chooses which VM should get ownership of the device.

Entry

EAX = VM handle of current device owner
EBX = VM handle of contending VM (Must be **Cur_VM_Handle**)
ESI -> 8 byte device name SPACE PADDED!!!

Exit

EBX = VM handle of contention winner
If carry is set then contention could not be resolved

Uses

EBX, Flags

SHELL_SYSMODAL_Message

Description

This procedure is called to put up SYSMODAL messages. Refer to SHELL.INC and the *Windows SDK* for information on message box parameters.

Entry

EBX = VM Handle of VM responsible for message
EAX = Message box flags (SEE MB_xxxx in SHELL.INC)
NOTE THAT MB_SYSTEMMODAL MUST BE SET.
ECX -> NUL terminated Message Text
EDI -> NUL terminated Caption Text == 0 for standard caption
-> NUL for No caption

Exit

EAX = Response code from dialog box (SEE IDxx in SHELL.INC)

Uses

Flags, **EAX**

Chapter 35 Virtual Display Device (VDD) Services

These are the Virtual Display Device (VDD) services. See Chapter 18, "The VDD and Grabber DLL," for a more detailed explanation.

35.1 Displaying a VM's Video Memory in a Window

There are several API services supplied to efficiently render a VM's video memory into a window. These routines are called by the Grabber. Since the Grabber runs in a virtual machine, parameters are passed in the Client Registers and in VM memory pointed to by the Client Registers.

The first step in updating windowed VMs is for the Shell to call **Set_VMState** with a parameter indicating that the VM is to be windowed. This will enable the VDD controller and memory state tracking and reporting of changes. When the VM is no longer windowed, **Set_VMState** is called again. When the VMState is not windowed, the **Get_Mod** call will always return no changes, and the video update message will never be generated.

The Grabber has to be assured that the call to get the video memory is consistent with the call to get the video state; for example, displaying a mode 3 VM in mode 10 is inconsistent. To support this, the VM will not run after a Get_Mod or Get_Mem call. The VM resumes only after a Free_Mem or UnLock_App call. This way the VM's state will not change during the process of window updating.

Notice that when a VM's video state changes, including controller state changes such as cursor movement and memory modification, the VDD will send WINOLDAPP a display update message. All the changes made to the video state will accumulate and be reported by **Get_Mod** until a **Clear_Mod** call is made. There will only be one display update message per **Clear_Mod** call.

VDD_Msg_BakColor

Description	After calling **Begin_Message_Mode**, this service sets up the background attribute.
Entry	**EAX** = Color (for EGA/VGA driver, a text mode attribute) **EBX** = VM handle
Exit	None

Uses Flags

VDD_Msg_ClrScrn

Description This routine is called by the Shell to initialize the screen for putting up messages. If the focus VM is the current VM, it will clear the screen immediately. Otherwise, the screen will be initialized when the focus changes. A **Begin_Message_Mode** device control must be issued before this service is used.

Entry
EBX = VM handle
EAX = background attribute

Exit
EAX = width in columns
EDX = height in rows

Uses Flags, **EAX**, **EDX**

VDD_Msg_ForColor

Description After calling **Begin_Message_Mode**, this service sets up the foreground attribute.

Entry
EAX = Color (for EGA/VGA driver, a text mode attribute)
EBX = VM handle

Exit None

Uses Flags

VDD_Msg_SetCursPos

Description After calling **Begin_Message_Mode**, this routine sets the cursor position.

Entry
EAX = row
EDX = column
EBX = VM handle

Exit None

Uses Flags

VDD_Msg_TextOut

Description After calling **Begin_Message_Mode** and setting up the foreground and background colors, this service puts characters on the screen.

Entry
ESI = address of string
ECX = length of string
EAX = row start
EDX = column start
EBX = VM handle

Exit None

Uses Flags

35.2 Miscellaneous VDD Services

The services discussed in this section provide other VDD functions not easily catagorized, such as hiding the cursor. They are provided here in alphabetical order.

VDD_Get_GrabRtn

Description This service returns the address of video grab routine. The grab routine is called by the Shell device when the appropriate hot key is pressed by the user. It makes a copy of the visible screen and controller state of the current VM. That copy is then accessible via the **GRB_Get_GrbState** and **GRB_Get_GrbMem** services.

Entry None

Exit **ESI** = address of grab routine

Uses Flags, **ESI**

VDD_Get_ModTime

Description This routine is used to determine if any video activity has occurred. The poll device uses it to determine if the VM is idle.

Entry EBX = VM handle

Exit EAX = System Timer at last video modification

Uses Flags, EAX

VDD_Get_Version

Description This service returns the version number and device ID.

Entry None

Exit ESI = ptr to 8 byte ID string
AH = major version
AL = minor version
Carry Flag clear

Uses Flags, AX, ESI

VDD_Hide_Cursor

Description This service hides/shows the cursor in a window. If **EAX** is nonzero, then this service sets a hide cursor flag or else clears the flag. This is so that, if the mouse is using a hardware cursor, it can turn off that cursor while the VM is windowed (since the VM will no longer own the mouse).

Entry EAX = 0 if cursor SHOULD be displayed in a window
!= 0 if cursor SHOULD NOT be displayed in a window
EBX = control block pointer

Exit None

Uses Flags

VDD_PIF_State

Description This service informs the VDD about PIF bits for the VM just created.

Entry **EBX** = VM handle
AX = PIF bits

Exit None

Uses Flags

VDD_Set_HCurTrk

Description This service sets flag passed to VMDOSAPP indicating that VMDOSAPP should maintain the cursor position within the display window for this application. This is called by the Keyboard driver when a keyboard interrupt is simulated into a VM.

Entry **EBX** = VM handle

Exit None

Uses Flags

VDD_Set_VMType

Description This service is used to inform the VDD of a VM's type. The parameter explicitly passed is the windowed flag. The VM status flags, Exclusive and Background, are implicity passed. This should be called prior to running the VM and each time thereafter that any of the VM parameters are modified. Notice that, for a system critical **Set_Focus**, this routine may not be called before the **Set_Focus**. In that case, the VDD is responsible for doing an implied **Set_VMType** (not windowed).

Entry **EAX** = state flag (= nonzero if changing to windowed VM)
EBX = VM handle whose state is to change

Exit None

Uses Flags

VDD_Query_Access

Description This service is used by the other virtual devices when they want to access video memory. The VxD should not access video memory unless this routine says it is OK.

Entry	**EBX** = VM handle
Exit	if access is OK, carry flag = 0 else carry flag = 1
Uses	Flags

Chapter 36 Virtual Keyboard Device (VKD) Services

The Virtual Keyboard Device (VKD) provides services that support hot keys, Message Mode key handling, and keyed input to VMs. The services are presented in the following order:

- **VKD_API_Force_Key**
- **VKD_API_Get_Version**
- **VKD_Cancel_Hot_Key_State**
- **VKD_Cancel_Paste**
- **VKD_Define_Hot_Key**
- **VKD_Define_Paste_Mode**
- **VKD_Flush_Msg_Key_Queue**
- **VKD_Force_Keys**
- **VKD_Get_Kbd_Owner**
- **VKD_Get_Msg_Key**
- **VKD_Get_Version**
- **VKD_Local_Disable_Hot_Key**
- **VKD_Local_Enable_Hot_Key**
- **VKD_Peek_Msg_Key**
- **VKD_Reflect_Hot_Key**
- **VKD_Remove_Hot_Key**
- **VKD_Start_Paste**

These are protected-mode API services used by WINOLDAP to send keys to a windowed VM.

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

VKD_API_Force_Key

Description This service forces a key into a VM as if it were typed on the keyboard. Because VKD will scan these forced keys for hot keys, forcing VKD hot keys is allowed.

Entry
EBX = VM handle (0 for current focus)
CH = scan code
CL = repeat count (1 or more)
EDX = shift state (-1 means no change)

Exit Carry Set, if error

Uses None

NOTE Currently limited to focus VM, so service will fail if EBX # 0 or EBX # focus VM handle.

VKD_API_Get_Version

Description This service gets the version number of the VKD device.

Entry None

Exit
AH = major, **AL** = minor
Carry clear

Uses None

VKD_Cancel_Hot_Key_State

Description This service causes the VKD to exit the hot key state.

Entry None

Exit Keys will start being passed into the focus VM again

Uses None

VKD_Cancel_Paste

Description This service cancels the paste that was started in the VM with **VKD_Start_Paste.**

Entry **EBX** is VM handle

Exit None

Uses Flags

VKD_Define_Hot_Key

Description This service defines a hot key notification routine. Hot keys are detected by ANDing the shift state mask with the global shift state, then comparing the resulting state with the shift state compare value. If this matches, and the key code matches, then the callback routine is called with the specified reference data in **EDX.**

Entry

AL = scan code of the main key
AH = 0, if normal code
AH = 1, if extended code (ExtendedKey_B)
AH = 0FFh, if either (AllowExtended_B)
EBX = shift state

high word is mask that is ANDed with the global shift state when checking for this hot key; low word is masked shift state compare value.

Equates for common shift mask and compare values are defined in VKD.INC:

HKSS_Shift for either shift key
HKSS_Ctrl for either control key
HKSS_Alt for either ALT key

The macro ShiftState is also defined to load **EBX** with the mask and compare value. e.g.,

ShiftState <SS_ALT + SS_Toggle_mask>, SS_RAlt

loads **EBX** so that the hot key will only be recognized when the Right ALT key is held down.

VKD>INC also defines "SS_" equates for the different shift state bits and common combinations of bits.

CL = flags

CallOnPress - Call callback when key press is detected
CallOnRelease - Call callback when key release is detected (keyboard may still be in hot-key hold state)
CallOnRepeat - Call callback when repeated press is

detected
CallOnComplete - Call callback when the hot key state is ended(all shift modifier keys are released) or when a different hot key is entered (i.e. pressing ALT 1 2, if both ALT+1 and ALT+2 are defined hot keys, then ALT+1's callback will be called before ALT+2's to indicate that the ALT+1 is complete even though the ALT key is still down)
CallOnUpDwn - Call on both press and release
CallOnAll - Call on press, release and repeats
PriorityNotify - Used with one of the call options to specify that the callback can only be called when interrupts are enabled and the critical section is un-owned
Local_Key - Key can be locally enabled/disabled

ESI = offset of callback routine
EDX = reference data
EDI = maximum notification delay if PriorityNotify is set, 0, means always notify (milliseconds)

Exit

```
If Carry clear then
     EAX = definition handle
else the definition failed (no more room)
```

Uses

Flags

Callback

Called when hot key is detected, and detection meets mask requirements. (CallOnPress, CallOnRelease, CallOnRepeat, CallOnUpDwn, or CallOnAll)

AL = scan code of key
AH = 0, if key just pressed (Hot_Key_Pressed)
= 1, if key just released (Hot_Key_Released)
= 2, if key is an auto-repeat press (Hot_Key_Repeated)
= 3, hot key state ended (Hot_Key_Completed)
EBX is hot key handle
ECX = global shift state
EDX is reference data
EDI = elapsed time for delayed notification (milliseconds)
(normally 0, but if PriorityNotify is specified then this value could be larger)
This procedure can modify **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, and Flags

VKD_Define_Paste_Mode

Description This service selects the VM's paste mode, whether INT 16 pasting can be attempted or not. Some applications hook INT 9 and do things that will not allow pasting to be done through INT 16H. Normally, VKD can detect this by setting a timeout to see if any INT 16s are being done by the application, and if not, then switching to INT 9 paste. But, some applications may do some INT 16s, in which case the paste would be broken. Therefore, this service is provided to allow the Shell device to force a VM into INT 9 paste, based only on a PIF bit.

Entry
AL = 0 allow INT 16 paste attempts
AL = 1 force INT 9 pasting
EBX = VM handle

Exit None

Uses Flags

VKD_Flush_Msg_Key_Queue

Description This service flushes any available keys from the special message mode input buffer.

Entry **EBX** = VM handle

Exit Input buffer has been cleared

Uses Flags

VKD_Force_Keys

Description This service forces scan codes into the keyboard buffer that look exactly like they had been typed on the physical keyboard. These keys will be processed in the context of the focus VM.

Entry
ESI points to a buffer of scan codes
ECX is # of scan codes in the buffer

Exit

```
If the keyboard buffer was overflowed, then
     Carry set
     ECX is # of remaining scan codes that did not fit
```

Uses ECX,Flags

VKD_Get_Kbd_Owner

Description This service gets the VM Handle of the keyboard focus VM.

Entry None

Exit **EBX** = VM Handle of keyboard owner

Uses Flags, **EBX**

VKD_Get_Msg_Key

Description This service returns the next available key from the special message mode input buffer and removes it from the buffer. If no key is available, then it returns with the Z flag set. (This is not a blocking read!)

Entry **EBX** = VM handle

Exit Z flag clear, if key was read
AL = scan code
AH = modifier flags
MK_Shift - a SHIFT key is down
MK_Ctrl - a CTRL key is down
MK_Alt - an ALT key is down
MK_Extended - the key is an extended key
Z flag set, if no key available

Uses **EAX**, Flags

VKD_Get_Version

Description This service gets the VKD version number.

Entry None

Exit **AH** = major, **AL** = minor
Carry Flag clear

Uses	**EAX**, Flags

VKD_Local_Disable_Hot_Key

Description	This service disables a hot key in the specified VM. It is only allowed on hot keys which were declared with the Local_Key bit set in CL.
Entry	**EAX** is hot key handle **EBX** is VM handle
Exit	None
Uses	Flags

VKD_Local_Enable_Hot_Key

Description	This service enables a hot key in the specified VM.
Entry	**EAX** is hot key handle **EBX** is VM handle
Exit	None
Uses	Flags

VKD_Peek_Msg_Key

Description	This service returns the next available key from the special message mode input buffer without removing it from the buffer. If no key is available, then it returns with the Z flag set.
Entry	**EBX** = VM handle
Exit	Z flag clear, if key available AL = scan code AH = modifier flags MK_Shift - a shift key is down MK_Ctrl - a control key is down MK_Alt - an alt key is down

MK_Extended - the key is an extended key
Z flag set, if no key available

Uses EAX, Flags

VKD_Reflect_Hot_Key

Description This service reflects a hot key into a specified VM and exits the hot key state. This service is normally called by a hot key notification callback routine. It enables the callback to send the hot key into a VM and pretend that it wasn't really recognized as a hot key. VKD will simulate the required key strokes to get the VM into the state of this specified shift state, then it will simulate the key strokes for the hot key itself, and finally simulate key strokes to get the VM to match the current global shift state.

Entry EAX is hot key handle
EBX is VM handle
CX is required shift state

Exit Hot key has been reflected, and VKD is no longer in hot key state

Uses Flags

VKD_Remove_Hot_Key

Description This service removes a defined hot key.

Entry EAX is hot key definition handle to be removed

Exit None

Uses Flags

VKD_Start_Paste

Description This service puts a VM into paste mode by simulating keyboard activity with keystrokes taken from the specified paste buffer. Depending on the mode set with the service **VKD_Define_Paste_Mode** (default is to try INT 16 pasting), VKD waits for the VM to poll the keyboard BIOS through its INT 16 interface. If the VM does keyboard input through the BIOS, then VKD will simulate the keyboard input at this high level (plugging in ASCII codes.) If the VM fails to perform any INT 16s within in a timeout period, or the mode has been set to avoid INT 16 pasting, then VKD will simulate the necessary hardware interrupts to perform the pasting. Physically typed hot keys are still processed while pasting is in progress.

Entry

EAX is linear address of paste buffer
the paste buffer contains an array of key structures:

OEM_ASCII_value	db	?
scan_code	db	?
shift_state	dw	?

shift state bits are:
0000000000000010b shift key depressed
0000000000000100b ctrl key depressed

The scan code should be FFh and the shift state FFFFh, if VKD should convert the key to a ALT+numpad sequence. (this information is identical to what is given by the Window's keyboard routine **OEMKeyScan**)

EBX is VM handle
ECX is number of paste entries in the paste buffer
ESI is call back address (can be 0)
EDX is reference data

Exit

Carry clear
paste is started
Carry set
paste failed, unable to allocate memory for buffer copy

Uses

Flags

Callback

Called when paste is completed or cancelled
EAX is completion flags
Paste_Complete - paste completed successfully
Paste_Aborted - paste cancelled by user
Paste_VM_Term - paste aborted because VM terminated
EBX is VM handle of VM that was receiving the paste
EDX is reference data
Procedure can modify **EAX, EBX, ECX, EDX, ESI, EDI,** and Flags

Chapter 37 Virtual PIC Device (VPICD) Services

The Virtual Programmable Interrupt Controller Device (VPICD) routes hardware interrupts to other virtual devices, provides services that allow virtual devices to request interrupts, and simulates hardware interrupts into virtual machines. See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general discussions of the VPICD.

Peripherals, such as disk drives and COM ports, use hardware (physical) interrupts to notify software of changes in their status.

The topics in this chapter are presented in the following order:

- Default Interrupt Handling
- Virtualizing an IRQ
- Virtualized IRQ Callback Procedures
- VPICD Services
- Grabber

37.1 Default Interrupt Handling

The most basic function of VPICD is to emulate the functions of the physical interrupt controller (PIC). This entails reflecting interrupts into virtual machines and simulating I/O such as recognizing when a VM issues an EOI (End Of Interrupt), reading the mask register, etc. When VPICD is initialized, it sets up a default interrupt handler for every Interrupt ReQuest (IRQ). These handlers determine which VM an interrupt should be reflected into, and they arbitrate conflicts between virtual machines that attempt to unmask the same interrupt.

An interrupt that is unmasked when enhanced Windows is initialized is considered a global interrupt. A global interrupt will always be reflected into the currently executing virtual machine, and any VM can mask or unmask the IRQ. If a virtual machine unmasks an IRQ that was masked when the enhanced Windows environment was initialized, it will own that IRQ. All interrupts for an owned IRQ will be reflected only to the IRQ's owner. If another virtual machine attempts to unmask the interrupt, the second VM will be terminated and the user will see a dialog box that tells him to reboot his computer.

It is important to remember that this is only the default behavior of VPICD. If another virtual device virtualizes an IRQ it is up to the device that virtualized the interrupt to determine which VMs receive interrupts and arbitrate conflicts. Once an IRQ is virtualized, VPICD's default handling for that IRQ stops.

37.2 Virtualizing an IRQ

When a virtual device needs to hook a specific IRQ (Interrupt ReQuest), it must ask VPICD for permission. If another device has already virtualized the IRQ, then the call will fail if either of the VxDs is unable to share the IRQ (both must have the **Can_Share** option set for two VxDs to use the same IRQ).

When a VxD calls **VPICD_Virtualize_IRQ**, it passes a pointer to a structure called an IRQ Descriptor that contains the number of the IRQ and the address of several callback procedures. This structure is included in the file VPICD.INC:

```
VPICD_IRQ_Descriptor  STRUC
        VID_IRQ_Number       dw   ?
        VID_Options          dw   0
        VID_Hw_Int_Proc      dd   ?
        VID_Virt_Int_Proc    dd   0
        VID_EOI_Proc         dd   0
        VID_Mask_Change_Proc dd   0
        VID_IRET_Proc        dd   0
        VID_IRET_Time_Out    dd   500
VPICD_IRQ_Descriptor  ENDS
```

The **VID_IRQ_Number** contains the number of the IRQ the VxD wishes to virtualize. **VID_Options** is a bit field that is used to specify special options. The next five fields specify the address of various callback procedures. The final field determines the maximum amount of time in milliseconds that VPICD will allow before the interrupt is timed-out. Time-outs are very important to prevent the enhanced Windows environment from hanging while simulating a hardware interrupt.

37.3 Virtualized IRQ Callback Procedures

A virtual device may specify up to five callback procedures in its **IRQ_Descriptor** structure. One of these, **Hw_Int_Proc**, is required. The other callback procedures are optional and are simply used to inform a virtual device whenever the state of the virtualized IRQ changes. For example, the **Virt_Int_Proc** procedure will be called whenever an interrupt is simulated into a VM; the **Mask_Change_Proc** is called whenever a virtual machine masks or unmasks the interrupt, etc. Each of the callback procedures is described in this section in detail and in alphabetical order. Callback procedures may modify **EAX, EBX, ECX, EDX, ESI,** and Flags. Although they will be called with interrupts disabled, they are allowed to enable them. If the procedures perform a lot of processing, interrupts should be executed.

VID_Hw_Int_Proc

Description

The **VID_Hw_Int_Proc** procedure is called whenever a hardware interrupt occurs. Notice that the procedure is just that, a procedure that returns using a near return — not an IRET. Since the the VxD environment kernel is single-threaded, the services that this procedure is allowed to call are limited because it is possible for an interrupt to occur while executing in the VMM. Therefore, many interrupt procedures will need to use the **Schedule_Call_Global_Event** services to perform additional processing of an interrupt. A typical **VID_Hw_Int_Proc** will service the physical device, call **VPICD_Phys_EOI** to end the physical interrupt, and set the virtual IRQ request for a specific virtual machine. Some devices may never request an interrupt for a virtual machine and others may request more than one interrupt per physical interrupt. In any case, every physical interrupt does not need to be reflected 1-1 into a virtual machine.

Entry

```
Interrupts Disabled
EAX = IRQ handle
EBX = Current VM handle
```

Exit

None

VID_EOI_Proc

Description

The **VID_EOI_Proc** callback is normally used for devices that are partially virtualized. For example, the Virtual Mouse Device (VMD) lets the MS-DOS mouse driver handle all I/O with the mouse hardware. The VMD just reflects the interrupt to the VM that owns the mouse. Since it doesn't service the device during the **VMD_Hw_Int** procedure, it can't call **VPICD_Phys_EOI** at this point (since it's not the end of the interrupt). Once a virtual machine has serviced the interrupt, it will issue an EOI and, at this point, the VMD calls **VPICD_Clear_Int_Request** followed by **VPICD_Phys_EOI**. The default interrupt routines need the **VID_EOI_Proc** callback for the same reason — they have to wait for the VM to service the interrupting device before they physically signal an EOI to the IRQ.

Entry

```
Interrupts Disabled
EAX = IRQ handle
EBX = Current VM handle
```

Exit

None

VID_Virt_Int_Proc

Description

The **VID_Virt_Int_Proc** callback can be useful for implementing critical sections around a simulated hardware interrupt. A VxD will request an interrupt, and that interrupt may be simulated at a later point in time. This callback is issued at the point when the interrupt is actually being simulated into the virtual machine. This call is made after the "point of no return" has been passed. Therefore, it is impossible for a virtual device to stop the interrupt once this call has been issued. A VxD that uses this callback will usually also use the **VID_Virt_IRET_Proc** callback to detect the end of the simulated interrupt.

Entry

```
Interrupts Disabled
EAX = IRQ handle
EBX = Current VM handle
```

Exit

None

VID_IRET_Proc

Description

This callback is useful for devices that must simulate large numbers of interrupts in a short period of time. For example, the Virtual COM Device will simulate an interrupt, allow one character to be read from the COM port, and wait for the virtual machine to IRET before putting more data into the virtual COM receive buffer. This is because many programs would crash if too many bytes of data were queued and shovelled into the virtual machine too quickly. The crash would occur because the program's stack would overflow. For example, assume that a terminal program has an interrupt routine that looks like this:

```
push     ax               ; (Push AX, DX is the
push     dx               ; minimum possible)
(Read a byte from the COM port)
mov      al, 20h          ; Non-Specific EOI
out      20h, al          ; EOI the PIC
sti                       ; Enable interrupts
(Do other stuff)
pop      dx
pop      ax
iret
```

This is a perfectly valid interrupt procedure and, in fact, it is very common in actual terminal programs. Now consider what would happen if the Virtual COM Device (VCD) had 500 bytes of data queued, and it did not use the **VID_IRET_Proc** callback. When the VM reads a byte of data, VCD puts the next byte of data into the receive buffer and request another interrupt. When the terminal program executes the STI instruction, VPICD immediately simulates another COM interrupt. This sequence of events is repeated 499 times, each time nesting an interrupt while in the terminal program's interrupt routine. The problem is that the IRET frame on the stack requires 6 bytes per interrupt, and the 2 pushed registers take up 4 more bytes for a total of 10 bytes per interrupt. Since we would nest 500 interrupts, 5K bytes of stack space would be required.

Since this is obviously unacceptable, VCD waits for the terminal program to IRET before simulating another interrupt. The Virtual Timer uses similar logic to prevent shoving too many timer interrupts into a virtual machine.

Entry

```
Interrupts Disabled
EAX = IRQ handle
EBX = Current VM handle
If carry is set then interrupt timed-out
```

Exit None

VID_Mask_Change_Proc

Description The **VID_Mask_Change_Proc** is often used to detect contention for a device. The default interrupt routines use this callback to detect conflicts with nonglobal interrupts.

Entry

```
Interrupts Disabled
EAX = IRQ handle
EBX = Current VM handle
ECX = 0 if VM is unmasking IRQ, != 0 if masking IRQ
```

Exit None

37.4 VPICD Services

This section presents descriptions of VPICD services in alphabetical order.

VPICD_Call_When_Hw_Int

Description You must call this procedure with interrupts *disabled*. This service enables other VxDs to be notified when every hardware interrupt occurs. It is intended to be used by the Virtual DMA Device (VDMAD) to detect when a DMA transfer is complete. However, any VxD can use this service. It should be noted though, that since your callback will be called for every hardware interrupt, it could have a major performance impact on systems with devices that interrupt frequently. Therefore, you should avoid using this service.

A callback installed by this service is responsible for chaining to the next handler in the interrupt filter chain, and it must preserve the **EBX** register for the next handler.

```
Sample_Hook_Init:
        pushfd
        cli
        mov     esi, OFFSET32 My_Int_Hook
        VxDcall VPICD_Call_When_Hw_Int
        popfd
        mov     [Next_Int_Hook_Addr], esi
        clc
        ret
```

```
My_Int_Hook:
        push    ebx
        (Do something useful here)
        pop     ebx
        jmp     [Next_Int_Hook_Addr]
```

Entry **ESI** -> Procedure to call

Exit **ESI** -> Procedure to chain to

Uses **ESI**, Flags

Callback **EBX = Cur_VM_Handle**

VPICD_Clear_Int_Request

Description This service resets an IRQ request that was previously set by a call to **VPICD_Set_Int_Request**. If the IRQ is being shared with another device, then this service may not reset the virtual request if another device has also set the virtual IRQ. However, the request will be cleared when all devices that have called **Set_Int_Request** call this service.

Entry **EAX** = IRQ handle
EBX = VM handle

Exit Virtual IRQ request is cleared

Uses Flags

VPICD_Convert_Handle_To_IRQ

Description This service returns the number of the IRQ for the IRQ handle in **EAX**.

Entry **EAX** = IRQ Handle

Exit **ESI** = IRQ Number

Uses **ESI**, Flags

VPICD_Convert_Int_To_IRQ

Description This service takes an interrupt vector number and returns the number of the IRQ that is mapped to that interrupt. For example, INT 8 will typically be converted to IRQ 0. However, VMs are allowed to remap the virtual PIC to any interrupt vector they wish. Therefore, devices should never make assumptions about to which interrupt vector a particular IRQ is mapped.

Entry **EAX** = Interrupt vector number

Exit

```
If carry is clear then
    EAX = IRQ number
else
    Interrupt vector not mapped to any IRQ
```

Uses None

VPICD_Convert_IRQ_To_Int

Description This service accepts an IRQ number and returns an interrupt vector number for a specified VM. For example, typically IRQ 0 will be converted to INT 8 on an IBM PC. However, VMs are allowed to remap the virtual PIC to any interrupt vector they wish. Therefore, devices should never make assumptions about to which interrupt vector a particular IRQ is mapped.

Entry **EAX** = IRQ number — NOT HANDLE!
EBX = VM handle

Exit **EAX** = Interrupt vector

Uses **EAX**, Flags

VPICD_Get_Complete_Status

Refer to **VPICD_Get_Status** for description.

VPICD_Get_IRQ_Complete_Status

Description This service is similar to **VPICD_Get_Complete_Status** except that it takes an IRQ number as a parameter instead of an IRQ handle. This is useful for devices to inspect an IRQ before attempting to virtualize it or for inspecting the state of another device's interrupt.

Also, since it indicates whether or not an IRQ has been virtualized already, it can be used by devices to prevent conflicts when more than one device may want to use an IRQ.

Entry EAX = IRQ number

Exit ECX = Status as described for **VPICD_Get_Complete_Status**

```
If the carry flag is set then
    The IRQ has been virtualized
else
    The IRQ has not been virtualized
```

Uses ECX, Flags

VPICD_Get_Status

Description These services return the status of a virtual IRQ for a specified VM. The status returned in **ECX** is defined by equates in the VPICD.INC file. **VPICD_Get_Status** will only return the Virtual **In_Service** and **IRET_Pending** status bits. **VPICD_Get_Complete_Status** will return with all status bits defined. The shorter version is supplied because it is much faster, and the status returned contains the most commonly used information.

Entry **EAX** = IRQ handle
EBX = VM handle

Exit **ECX** = Status flags (see equates VPICD.INI)

Bit	Description
0 = 1	A Virtual IRET is pending
1 = 1	The IRQ is virtually in service
2 = 1	The IRQ is physically masked
3 = 1	The IRQ is physically in service
4 = 1	VM has masked the IRQ
5 = 1	The Virtual IRQ is set (by any VxD)
6 = 1	The physical IRQ is set
7 = 1	Tha calling VxD's Virtual IRQ is set

Uses ECX, Flags

VPICD_Get_Version

Description This service returns the VPICD major and minor version numbers.

Entry None

Exit
AH = Major version
AL = Minor version
EBX = Flags
Bit 0 = 1 - Master/Slave PC/AT type configuration
0 - PC/XT type single PIC configuration
Other bits reserved for future versions.
ECX = Maximum IRQ supported (07H or 0FH)
Carry flag clear

Uses **EAX, EBX, ECX,** Flags

VPICD_Phys_EOI

Description Calling this procedure will end a physical interrupt and will allow further hardware interrupts from the specified IRQ. Notice that an interrupt that is physically in service will *not* suppress interrupts to "lower priority" IRQs, since VPICD does not prioritize hardware interrupts. Therefore, it is acceptable for an interrupt to be physically in service for an arbitrary length of time.

Entry **EAX** = IRQ handle

Exit None

Uses Flags

VPICD_Physically_Mask

Description This service will mask the specified IRQ on the hardware PIC. This will suppress all hardware interrupts on the IRQ until **VPICD_Physically_Unmask** or **VPICD_Set_Auto_Masking** is called.

Entry **EAX** = IRQ handle

Exit IRQ is masked

Uses Flags

VPICD_Physically_Unmask

Description This service will unmask the specified IRQ on the hardware PIC regardless of the mask state of virtual machines. This means that even if every VM has masked the virtual IRQ, the physical IRQ will remain unmasked.

Entry **EAX** = IRQ handle

Exit IRQ is masked

Uses Flags

VPICD_Set_Auto_Masking

Description Automatic masking is the default state for every IRQ. It can be overridden by **VPICD_Physically_Mask/Unmask.** When automatic masking is used, the state of the physical mask is determined by the state of every virtual machine's virtual mask. If at least one VM has the IRQ unmasked, then the physical IRQ will remain unmasked. Otherwise, the IRQ will be masked on the hardware PIC.

Entry **EAX** = IRQ handle

Exit IRQ will be physically unmasked if at least one VM has unmasked the IRQ.

Uses Flags

VPICD_Set_Int_Request

Description This service sets the virtual interrupt request for the specified IRQ and VM. It may cause an interrupt to be simulated immediately. However, in many cases, the interrupt will *not* be simulated until a later point in time. The interrupt will not be simulated immediately if:

- The virtual machine has interrupts disabled.
- The virtual machine has masked the IRQ.

- A higher priority virtual IRQ is in service.
- It is not possible to run the specified VM (it is suspended, etc).
- There are other reasons the interrupt may be postponed.

However, since the interrupt may be simulated immediately, virtual devices that have a virtual interrupt handler must be able to handle the case when their virtual interrupt procedure is called before this service returns.

Setting an interrupt request is not a guarantee that the interrupt will ever be simulated. For example, if the VM has masked the interrupt and never unmasks it, the interrupt will never be simulated. Also, a call to **VPICD_Clear_Int_Request** that is made before the virtual interrupt is simulated will prevent the interrupt simulation.

It is important to keep in mind that VPICD simulates a level triggered PIC. This means that once a virtual EOI occurs, another interrupt will be simulated immediately unless the virtual interrupt request is cleared.

Entry

EAX = IRQ handle
EBX = VM handle

Exit

Virtual IRQ request is set

Uses

Flags

VPICD_Test_Phys_Request

Description

This service will return with Carry set if the physical (hardware PIC) interrupt request is set for the specified IRQ.

Entry

EAX = IRQ handle

Exit

Carry flag = Physical Interrupt Request state

Uses

Flags

VPICD_Virtualize_IRQ

Description

This is not an async service; it cannot be called during an interrupt. This service is used to gain access to a specified virtual interrupt request. The caller passes this procedure a pointer to the IRQ descriptor (the structure declared in VPICD.INC) which specifies:

- IRQ number (required)
- Options
- Hardware interrupt handler (required)
- Virtual interrupt handler
- Virtual EOI handler
- Virtual mask change handler
- Virtual IRET handler
- Virtual IRET time-out (0 for no time-out)

For more information on the various options and parameters to this service see Section 37.3 "Virtualizing an IRQ," earlier in this chapter. When this service returns, if Carry is set, then the IRQ cannot be virtualized. Otherwise, **EAX** contains an IRQ handle. This handle is used for all subsequent communication with VPICD.

If every device that virtualizes the IRQ has the **Can_Share** option set then the IRQ can be shared by up to 32 devices.

Entry **EDI -> VPICD_IRQ_Descriptor**

Exit

```
If carry clear then
    EAX = IRQ Handle
else
    Error - Handle already allocated or invalid IRQ #
```

Uses **EAX** , Flags

Chapter 38 Virtual Sound Device (VSD) Services

These two services enable VxDs to generate a warning beep or return the VSD version number:

- **VSD_Bell**
- **VSD_Get_Version**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

VSD_Bell

Description	This service is provided so that devices can generate a warning beep. This is normally used when the user presses an invalid key or when an error occurs. Notice that this service will produce a 1/2-second tone, but it will then return immediately (it does not busy wait).
Entry	None
Exit	None
Uses	Flags

VSD_Get_Version

Description	This service returns the version number of the Virtual Sound Device.
Entry	None
Exit	AH = Major version number AL = Minor version number Carry flag clear
Uses	EAX, Flags

Chapter 39 Virtual Timer Device (VTD) Services

This chapter presents descriptions of the following VTD services:

- **VTD_Begin_Min_Int_Period**
- **VTD_Disable_Trapping**
- **VTD_Enable_Trapping**
- **VTD_End_Min_Int_Period**
- **VTD_Get_Interrupt_Rate**
- **VTD_Get_Version**
- **VTD_Update_System_Clock**

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions.

VTD_Begin_Min_Int_Period

Description

This service is used by VxDs to ensure a minimum accuracy for system timing. When this service is called, if the interrupt period specified is lower than the current timer interrupt period, the interrupt period will be set to the new frequency.

Until a matching **VTD_End_Min_Int_Period** call is made, the timer interrupt period is guaranteed never to be slower than the value specified.

A VxD should call this service only once before calling **VTD_End_Min_Int_Period.**

Typically the **Begin/End_Min_Int_Period** services are used by devices such as execution profilers that need extremely accurate timing. VMM system time-out services rely on the VTD to keep time. Therefore, more frequent timer interrupts will allow the time out services to be more accurate.

WARNING Fast timer interrupt periods can be very, very expensive in terms of total system performance. For example, on some machines a timer interrupt of 1 millisecond will degrade total machine throughput by 10 percent and disk I/O by up to 50 percent.

Entry EAX = Desired interrupt period

Exit

```
If carry clear then
    Interrupt period set
else
    Specified interrupt period is not valid
```

Uses Flags

VTD_Disable_Trapping

Description This service will force VTD to stop I/O trapping on the timer ports for a specified virtual machine. **VTD_Enable_Trapping** must be called once for every call made to this service. By default, timer port trapping is enabled when a VM is created.

It is sometimes necessary to disable temporarily I/O trapping for virtual machine code that reads the timer in extremely tight timing loops. A good example is the hard disk BIOS code that reads the ports hundreds of times per disk transfer. The overhead for servicing the I/O traps would cause disk performance to slow to a crawl.

WARNING This service must be used very carefully. If a VM reprograms the timer while port trapping is disabled, system timing will behave randomly. Only "trusted" code should be executed when timer port trapping is disabled.

If this service is called N times, then **VTD_Enable_Trapping** must also be called N times before trapping is reenabled. This allows nested calls to this service by more than one VxD.

Entry EBX = VM handle

Exit None

Uses Flags

VTD_Enable_Trapping

Description This service must be called to re-enable timer I/O port trapping after calling **VTD_Disable_Trapping**. Notice that this call must be made once for every call to **VTD_Disable_Trapping**. Only when every disable call has been matched by a call to this service will port trapping be reenabled.

Entry **EBX** = VM handle

Exit None

Uses Flags

VTD_End_Min_Int_Period

Description This service allows a device to "unrequest" a timer interrupt period that it set earlier through the **VTD_Begin_Min_Int_Period** service. See the documentation for **VTD_Begin_Min_Int_Period** earlier in this chapter for more information on the proper use of this service.

Entry **EAX** = Value passed earlier to **Begin_Begin_Min_Int_Period**

Exit

```
If carry clear then
    Interrupt period request removed successfully
else
    Specified interrupt period is not valid
```

Uses Flags

VTD_Get_Interrupt_Period

Description This service returns the current timer interrupt period.

Entry None

Exit **EAX** = Length of time between ticks in milliseconds

Uses Flags

VTD_Get_Version

Description This service returns the version number and the range of interrupt periods allowable by this device.

Entry None

Exit **EAX** = Version number (AH = Major, AL = Minor)
EBX = Fastest possible interrupt period in milliseconds
ECX = Slowest possible interrupt period in milliseconds
Carry flag clear

Uses **EAX, EBX, ECX,** Flags

VTD_Update_System_Clock

Description This service should *only* be called by the VMM. Devices should call the **Get_System_Time** VMM service. The VMM will then call this service to update the system clock.

Entry None

Exit None

Uses Flags

Chapter 40 V86 Mode Memory Manager Device Services

The V86MMGR is responsible for managing memory in the Virtual 8086 portion of each VM. It supports EMS and XMS, is responsible for allocating the base memory for VMs when they are created, and translates APIs from protected-mode applications into V86 calls for other VxDs.

See Chapter 16, "Overview of Windows in 386 Enhanced Mode," and Chapter 17, "Virtual Device Programming Topics," for general environment discussions. Other chapters that discuss memory management are Chapter 19, "Memory Management Services," and Chapter 6, "Network Support," in the *Microsoft Windows Device Driver Adaptation Guide*. Memory management is also discussed in the *Microsoft Software Development Kit, Programming Tools.*

The V86MMGR services are presented as follows:

- Initialization Services

  V886MMGR_Get_Version

  V86MMGR_Allocate_V86_Pages

  V86MMGR_Set_EMS_XMS_Limits

  V86MMGR_Get_EMS_XMS_Limits

- API Translation and Mapping Services

  V886MMGR_Set_Mapping_Info

  V86MMGR_Xlat_API

  V86MMGR_Load_Client_Ptr

  V86MMGR_Allocate_Buffer

  V886MMGR_Free_Buffer

  V86MMGR_Get_Xlat_Buff_State

  V86MMGR_Set_Xlat_Buff_State

  V86MMGR_Get_VM_Flat_Sel

  V86MMGR_Get_Mapping_Info

  V86MMGR_Map_Pages

  V86MMGR_Free_Page_Map_Region

40.1 Initialization Services

These services are used when a VM is created except for the **V86MMGR_Get_Version**, which may be used anytime.

V86MMGR_Get_Version

Description Returns the version of the V86MMGR VxD.

Entry None

Exit **AH** = Major version number
AL = Minor version number
Carry flag clear

Uses EAX, Flags

V86MMGR_Allocate_V86_Pages

Description This service is used by the SHELL VxD to set up the initial base memory of a VM when it is created. It allocates the memory, maps it into the virtual machine, and does a local **Assign_Device_V86_Pages** for the region allocated.

Entry **EBX** = VM handle
ESI = Desired size of VM address space in K bytes
EDI = Minimum size of VM address space in K bytes
ECX = Flags, see bit definitions in V86MMGR.INC

NOTE The **ESI** and **EDI** sizes include the **0-First_VM_Page** region of V86 address space.

Exit If carry set then
    ERROR: Could not allocate memory
else
    Memory allocated and mapped into VM
EAX = ACTUAL number of pages allocated and mapped (size of VM). Notice that this size *does not* include the space from 0-**First_VM_Page**

Uses EAX,Flags

V86MMGR_Set_EMS_XMS_Limits

Description This service is used by the SHELL VxD to set the EMS and XMS limit parameters for a VM.

Entry
EBX = VM handle to set limits of
EAX = Min EMS kilobytes
EDX = Max EMS kilobytes
ESI = Min XMS kilobytes
EDI = Max XMS kilobytes
ECX = Flag bits, see V86MMGR.INC

Notes
To disable access to XMS or EMS memory, set Max = Min = 0
To set only *one* of the two limits, set the OTHER Max = Min = -1
The XMS Limit *does not* include the HMA.

Exit
If carry set then could not set limits
Insufficient memory for Min allocation request
note that some of the limits may have been set. To Find out what happened, use **V86MMGR_Get_EMS_XMS_Limits**
else limits set

Uses Flags

V86MMGR_Get_EMS_XMS_Limits

Description This service is used by the SHELL VxD to get the EMS and XMS limit parameters for a VM.

Entry EBX = VM handle to get limits of

Exit
EAX = Min EMS kilobytes (always a multiple of 4)
EDX = Max EMS kilobytes (always a multiple of 4)
ESI = Min XMS kilobytes (always a multiple of 4)
EDI = Max XMS kilobytes (always a multiple of 4)
ECX = 0 if access to the HMA is disabled
ECX = 1 if access to the HMA is enabled

Uses EAX, ECX, EDX, ESI, EDI, Flags

40.2 API Translation and Mapping

One of the major roles of the V86MMGR is to provide a mechanism for other VxDs to translate API calls made from application software running in protected mode into the V86 portion of the virtual machine. The term "API translation" is used in this document to describe the conversion of an API call in protected mode into a corresponding V86 mode call. Because enhanced Windows runs under a standard DOS, DOS and BIOS calls must be reflected to V86 mode code to handle the call. There is a layer of code in the DOSMGR device that converts protected mode DOS calls into V86 calls.

The main translation service, **V86MMGR_Xlat_API**, is a simple interpreter that copies data into a buffer in the V86 address space and converts pointers to point to the copied data. Note that the data is *copied*. The memory is not mapped into V86 memory by changing page tables.

Other services are provided to allocate buffer space, map memory into global V86 address space, and perform other functions necessary for API translation.

Note that the translation services only work for the current VM and most must be called when running in the protected mode portion of the VM.

40.2.1 Basic API Translation

Many APIs require little or no translation. Others are extremely complex and require a great deal of coding. The simplest API is one that has no pointers. A software interrupt based API, in which all parameters are passed in the **EAX, EBX, ECX, EDX, ESI, EDI,** and **EBP** registers and flags, requires no special translation software. By default, enhanced Windows will reflect an interrupt that is executed in protected mode into V86 mode. For example, the BIOS printer interface (Int 17h) requires no translation code since all APIs are register-based with no pointers.

However, most APIs have at least some calls that take pointers as parameters. For example, to open a file through DOS, you must point at the name of the file to open with the **DS:DX** registers. Since the address that a protected mode program will pass in **DS:DX** is not usually addressable in the V86 portion of the VM, there must be code that copies the filename into a buffer that is addressable in V86 mode so that DOS can access the filename.

40.2.2 Complex API Translation

Some APIs are too complex or their buffers are too large to be handled by the **V86MMGR_Xlat_API** service. The DOS Exec function takes a pointer to a data structure that contains more pointers. This API requires special code to translate the pointers in the data structure and to copy the data that those pointers point to into V86 mode memory.

The DOS read and write file functions can have buffers as large as 64K. The typical V86MMGR translation copy buffer is 4K. Therefore, these calls require code to divide the call into several smaller reads or writes in V86 mode.

40.2.3 Hooking The Interrupt

Since the translation code should be the last protected mode handler you will need to hook the PM interrupt vector (using the **Hook_PM_Int_Vector** service) during the **Sys_Critical_Init** or **Device_Init** phases of initialization. All translation code should be initialized before the Init_Complete phase of initialization so that the **Exec_VxD_Int** service (provided by the VMM) can be used during this phase. Note that the V86MMGR translation services (except for **Set_Mapping_Info**) should not be called during **Sys_Critical_Init** or **Device_Init**.

By hooking the interrupt vector instead of using the **Hook_PM_Int_Chain** service you will allow protected mode applications to hook software interrupts "in front" of your translation code. This is very important for the Windows kernel since it needs to monitor the activity of Windows applications' API calls.

Sample Code

The code for a typical translation VxD looks like this:

```
VxD_ICODE_SEG
BeginProc My_Xlat_Init
        mov     eax, My_Translation_Int_Number
        VMMcall Get_PM_Int_Vector
        mov     [Chain_Segment], cx
        mov     [Chain_Offset], edx
        mov     esi, OFFSET32 My_Xlat_Procedure
        VMMcall Allocate_PM_Call_Back
        mov     ecx, eax
        movzx   edx, cx
        shr     ecx, 16
        mov     eax, My_Translation_Int_Number
        VMMcall Set_PM_Int_Vector
        clc
        ret
EndProc My_Xlat_Init
VxD_ICODE_ENDS

VxD_CODE_SEG
BeginProc My_Xlat_Procedure
        movzx   eax, [ebp.Client_AH]
        cmp     eax, My_Max_API_Number
        ja      Chain_To_Next_Handler
        VMMcall Simulate_Iret
        mov     edx, My_Trans_Script_Table[eax*4]
        VxDcall V86MMGR_Xlat_API
        ret
Chain_To_Next_Handler:
        movzx   ecx, [Chain_Segment]
        jecxz   Reflect_To_V86_Now
        mov     edx, [Chain_Offset]
        VMMcall Simulate_Far_Jmp
```

```
        ret
Reflect_To_V86_Now:
        VMMcall Begin_Nest_V86_Exec
        mov     eax, My_Translation_Int_Number
        VMMcall Exec_Int
        VMMcall End_Nest_Exec
        ret
EndProc My_Xlat_Procedure
VxD_CODE_ENDS
```

If the value in AH is not translated by this handler then it will be reflected to the next protected mode interrupt handler. If there is not another PM interrupt handler (code segment is zero) then the interrupt is immediately reflected to V86 mode.

You will note that My_Xlat_Procedure calls the **Simulate_Iret** service before it calls **V86MMGR_Xlat_API**. If you plan to "eat" an interrupt it is usually best to call this service first. If the iret was simulated after the call to **V86MMGR_Xlat_API** then any flags returned by the V86 interrupt handler would be destroyed (an iret pops flags from the interrupt stack frame).

40.2.4 Mapping vs. Copying

Some VxDs need to use the paging mechanism of the 386 to map pages from extended address space into the 1MB V86 address space of every virtual machine. The Virtual NetBIOS Device uses the mapping services when an asynchronous receive is issued so that the proper physical memory will be updated regardless of which VM is currently running. When memory is mapped using **V86MMGR_Map_Pages** it will be mapped to the same linear address in every virtual machine. Thus it is best to avoid using these services.

Do not use mapping as an alternative to copying just because you think mapping seems easier. It is faster to copy memory than to map it since the memory manager does not need to perform any page table mapping and locking. Mapping also uses a lot of address space (although it requires no memory). The mapping services should only be used for APIs that require memory mapped to the same address in every VM.

Note that the mapping services allow memory from one VM's V86 address space to be mapped into all VMs at a common address. *Don't use this for interprocess communication.* It will eat mapping space that may be required by other devices. If you want to design an IPC interface, either make it work for PM applications (which can share memory) or copy the data.

40.2.5 Writing Your Own Translation Procedures

Often, it is impossible to translate part or all of an API using the supplied macro interpreter. Therefore you may need to write procedures that do all or part of the translation. Examples of calls that require extra code are the DOS read and write commands and the get and set interrupt vector commands. The DOS commands to get and set interrupt vectors behave differently in protected mode since they must hook the protected mode interrupt vectors. These calls are never reflected to the "real" DOS running in V86 mode.

The DOS read and write file commands can use a buffer as large as 64K. Since the translation buffers can be as small as 4K, reads and writes must be divided before being reflected to DOS.

Since most APIs have some interfaces that can be handled by the **V86MMGR_Xlat_API** script language and others that must be translated by custom procedures you will probably want to dispatch to the custom procedures using the **Xlat_API_Jmp_To_Proc** macro.

To adjust V86 segment registers you should leave the VM in **PM_Exec_Mode** and change the **Alt_Client** registers. When in **PM_Exec_Mode** these registers contain the V86 segment registers and stack pointer. They will contain the PM segment registers and stack pointer when the VM is in **V86_Exec_Mode**.

40.2.6 Sample API Translation

This sample API is for an imaginary, incredibly simple network. The functions allow you to connect to a server and send or receive data. Assume that the network supports the following API from software interrupt 92h:

Function 0: Get version

Entry **AH** = 0

Exit **AH** = Major version
AL = Minor version

Function 1: Get Server Name

Entry **AH** = 1
DS:DX = Pointer to a 16 byte buffer to hold name

Exit None

Function 2: Connect To New Server

Entry **AH** = 2
DS:DX = Pointer to null terminated string that is name of server

Exit None

Function 3: Read/Write Data

Entry

AH = 3
ES:BX = Pointer to command block with following structure:

Offset	Size	Description
0	1	Command
1	2	Buffer size
3	4	Buffer pointer

Command field values:
0 = Read data from server
1 = Write data to server

Exit

None

Since function 0 is register based it requires no translation other than reflecting the interrupt to V86 mode. Functions 1 and 2 both can be translated by scripts using the **V86MMGR_Xlat_API** service. Function 3 requires a custom translation procedure.

```
VxD_DATA_SEG
Fctn_0_Script:
        Xlat_API_Exec_Int  92h
Fctn_1_Script:  Xlat_API_Fixed_Len ds, dx, 16
        Xlat_API_Exec_Int  92h
Fctn_2_Script:  Xlat_API_ASCIIZ   ds, dx
        Xlat_API_Exec_Int 92h
Fctn_3_Script:
        Xlat_API_Jmp_To_Proc Trans_Fctn_3
Copy_Command_Block_Script:
        Xlat_API_Fixed_Len es, bx, 7
        Xlat_API_Exec_Int  92h

Xlat_Ptr_Table:
        dd      OFFSET32 Fctn_0_Script
        dd      OFFSET32 Fctn_1_Script  dd      OFFSET32 Fctn_2_Script
        dd      OFFSET32 Fctn_3_Script
VxD_DATA_ENDS

VxD_CODE_SEG
BeginProc Translate_Sample_API
        movzx   edx, [ebp.Client_AH]
        cmp     edx, 3
        ja      Chain_To_Next_Handler
        VMMcall Simulate_Iret
        mov     edx, Xlat_Ptr_Table[edx*4]
        VxDcall V86MMGR_Xlat_API
```

```
        jc      Translation_Error       ret
Chain_To_Next_Handler:
        movzx   ecx, [Chain_Segment]
        jecxz   Reflect_To_V86_Now
        mov     edx, [Chain_Offset]
        VMMcall Simulate_Far_Jmp
        ret
Reflect_To_V86_Now:
        VMMcall Begin_Nest_V86_Exec
        mov     eax, 92h
        VMMcall Exec_Int VMMcall
        End_Nest_Exec
        ret
Translation_Error:
        Debug_Out "Unable to translate sample API"
        VMMjmp  Crash_Cur_VM
EndProc Translate_Sample_API

BeginProc Trans_Fctn_3
        push    fs
        push    gs
        pushad
; Get pointer to command block
        mov     ax, (Client_ES*100h)+Client_BX
        VxDcall V86MMGR_Load_Client_Ptr
; If command is invalid then fail the call
        mov     al, BYTE PTR fs:[esi]
        cmp     al, 1
        ja      Can_Not_Translate
; Get buffer size and pointer from command block
        mov     dx, fs
        mov     gs, dx
        mov edx, esi movzx ecx, WORD PTR gs:[edx+1]
        mov     fs, WORD PTR gs:[edx+5]
        movzx   esi, WORD PTR gs:[edx+3]
; Allocate a buffer, copying data if command is a write
        bt      eax, 0
        VxDcall V86MMGR_Allocate_Buffer
        jc      Can_Not_Translate
        mov DWORD PTR gs:[edx+3], edi
; Copy the command block and execute the interrupt
        push    edx
        mov     edx, OFFSET32 Copy_Command_Block_Script
        VxDcall V86MMGR_Xlat_API
        pop     edx
        jc      Can_Not_Translate
; Free the buffer, copying data if command is a read
        mov     al, BYTE PTR gs:[edx]
        bt      eax, 0
        cmc
        VxDcall V86MMGR_Free_Buffer
; Restore original pointer in command block
        mov     WORD PTR gs:[edx+5], fs
```

```
        mov     WORD PTR gs:[edx+3], si
        clc
Trans_F3_Exit:
        popad
        pop     gs
        pop     fs
        ret
Can_Not_Translate:      stc
        jmp     Trans_F3_Exit
EndProc Trans_Fctn_3

VxD_CODE_ENDS
```

V86MMGR_Set_Mapping_Info

Description

This service must be called during the **Sys_Critical_Init** or **Device_Init** phase of device initialization. It is used to define the minimum amount of translation buffer and global V86 map address space that will be required. VxDs such as the VNETBIOS use this service to ensure that there will be adequate global page mapping space to map network buffers. By default the translation copy buffer size is 4K and there are no global mapping pages.

Multiple VxDs may call this service. The V86MMGR will use the largest value for each of the parameters when allocating buffer space. In other words, if 10 VxDs request a two-page copy buffer then the copy buffer will be two pages (not 20).

Note that while a large copy buffer can speed up operations such as DOS reads, it requires extra memory to be allocated for every VM. Therefore, you should try to get by with a copy buffer size of one page if possible.

Entry

AL = Minimum number of pages required for default copy buffer
AH = Maximum number of pages desired for default copy buffer
BL = Minimum number of pages required for global page mapping region
BH = Maximum number of pages desired for global page mapping region

Exit

None

Uses

Flags

V86MMGR_Xlat_API

Description

This service is actually a simple interpreter that executes scripts that are created using macros defined in V86MMGR.INC. The macros are described in detail below.

Entry

EBX = Current VM handle
EBP -> Client register structure
EDX -> Script to translate

Exit

EDX is destroyed
If carry set then
    Error while executing script
else
    Script has been executed successfully

Uses

EDX, Flags

Xlat_API_Exec_Int [Int Number]

Terminates the interpretation of the translation script and reflects the specified interrupt into Virtual 8086 mode. When the interrupt returns then it will return to the caller.

```
DOS_No_Xlat_API:
        Xlat_API_Exec_Int 21h
```

Xlat_API_Fixed_Len [Segment], [Offset], [Length Constant]

Copies a fixed length buffer from extended memory into the translation buffer and fixes up the V86 Seg:Offset.

This service will fail if there is not enough room in the translation buffer to copy the data.

For example, the DOS Get Current Directory function (**AH**=47h), must be called with **DS:SI** pointing to a 64-byte buffer. The following script would perform the appropriate translation:

```
DOS_Get_Current_Directory_API:
        Xlat_API_Fixed_Len ds, si, 64
        Xlat_API_Exec_Int  21h
```

Xlat_API_Var_Len [Segment], [Offset], [Length Register]

Copies a variable number of bytes from extended memory into the translation buffer. This is used for APIs where the caller places the buffer size in a register.

This service will fail if there is not enough room in the translation buffer to copy the data.

For example, the Int 10h write string function (AH=0Eh), must be called with **ES:BP** pointing to the string to print and **CX** equal to the number of bytes to display. The following script would translate this call:

```
Int_10h_Write_String:
        Xlat_API_Var_Len  es, bp, cx
        Xlat_API_Exec_Int 10h
```

Xlat_API_Calc_Len [Segment], [Ptr_Off], [Calc_Proc_Addr]

Used to copy buffers that change in size. You must specify the selector:offset register pair that points to the buffer and the name of a procedure that will calculate the actual buffer size. The procedure will be called with **FS:ESI** pointing to the buffer and must return with **ECX** equal to the number of bytes to copy. The procedure must preserve all registers except **ECX**.

This service will fail if there is not enough room in the translation buffer to copy the data.

For example, the DOS buffered keyboard input command (AH=0Ah) can have a buffer size from 3 to 257 bytes long. The first byte of the buffer specifies the length of the input buffer as follows:

Byte	Contents
0	Maximum number of characters to read (1-255); this value must be set by the process before Function 0Ah is called.
1	Count of characters read.
2-(n+2)	Actual string of characters read, including the carriage return; n = number of bytes read.

The translation code for this API would look something like this:

```
VxD_DATA_SEG
Buff_Keyboard_Input_API:
        Xlat_API_Calc_Len ds, dx, Calc_Input_Buff_Size
        Xlat_API_Exec_Int 21h
VxD_DATA_ENDS

VxD_CODE_SEG
BeginProc Int_21_PM_To_V86_Translator

        cmp     [ebp.Client_AH], 0Ah
        jne     Not_Buffered_Keyboard_Input
        VMMcall Simulate_Iret
        mov     edx, OFFSET32 Buff_Keyboard_Input_API
        VxDcall V86MMGR_Xlat_API
        ret

EndProc Int_21_PM_To_V86_Translator

BeginProc Calc_Input_Buff_Size

        movzx   ecx, BYTE PTR fs:[esi]
        add     ecx, 2
        ret

EndProc Calc_Input_Buff_Size
VxD_CODE_ENDS
```

Xlat_API_ASCIIZ [Ptr_Seg], [Ptr_Off]

Copies a null-terminated string into V86 memory and adjusts the V86 pointer appropriately. Note that the string will not be copied back after the call is complete.

This service will fail if there is not enough room in the translation buffer to copy the string.

For example, the DOS Open File With Handle function (AH=3Dh), must be called with **DS:DX** pointing to the name of the file to open. The following script could be used translate the API:

```
DOS_Open_File_With_Handle:
        Xlat_API_ASCIIZ   ds, dx
        Xlat_API_Exec_Int 21h
```

Xlat_API_Jmp_To_Proc [Proc_Name]

Terminates the interpretation of the translation script and transfers control to a user defined procedure. The procedure can completely handle the API translation or can call **V86MMGR_Xlat_API** again. This can be useful for APIs that have several sub-APIs such as the DOS IOCTL calls.

The procedure will be called with **EBX** equal to Current VM Handle, **EBP** pointing to Client register structure, and **EDX** points to the next entry in the translation script (if there is one). It must preserve every register except for **EDX**. Therefore the procedure must preserve **EAX, EBX, ECX, ESI, EDI, EBP, DS, ES, FS,** and **GS.**

Your procedure should return with the carry flag clear if the translation was successful. Otherwise, it should return with carry set to indicate an error.

Xlat_API_Return_Ptr [Ptr_Seg], [Ptr_Off]

Used for calls that return a pointer to a structure. For 16-bit protected mode programs, if an appropriate selector does not exist to map the call, then this service automatically creates one. For 32-bit protected mode programs the selector returned will always be the **V86MMGR_VM_Flat_Selector** and the offset will be adjusted. Note that although this macro is placed before the **Exec_Int** macro in a translation script, the pointer is created after the interrupt has been executed.

This service will fail if it can not create an appropriate LDT selector.

For example, this service is used to translate Int 15h with AH=C0h, which returns a pointer in **ES:BX** that points to a hardware information structure on PS/2 machines. The following script would return the appropriate pointer:

```
Get_Machine_Info:
        Xlat_API_Return_Ptr es, bx
        Xlat_API_Exec_Int   15h
```

Xlat_API_Return_Seg [Ptr_Seg]

Used for calls that return a segment. If an appropriate selector does not exist to map the call then this service automatically creates one. Note that although this macro is placed before the **Exec_Int** macro in a translation script, the selector is created after the interrupt has been executed.

This service will fail if it can not create an appropriate LDT selector.

For example, this service is used to translate Int 15h with **AH=C1h**, which returns the segment of the EBIOS data area in **ES**. The following script would return a selector that points to the EBIOS data area:

```
Get_EBIOS_Selector:
        Xlat_API_Return_Seg es
        Xlat_API_Exec_Int   15h
```

Translating Multiple Pointers

The interpreter can copy multiple buffers. For example, the following translation table translates the DOS rename file call (AH = 56h):

```
Rename_API:
        Xlat_API_ASCIIZ   ds, dx
        Xlat_API_ASCIIZ   es, di
        Xlat_API_Exec_Int 21h
```

The first instruction copies the null-terminated string (ASCIIZ string) that **DS:DX** points to into the translation buffer in V86 memory, sets the V86 DS to the translation buffer segment, and changes **DX** to the offset in the buffer.

The second macro copies the ASCIIZ string that is pointed to by **ES:(E)SI** into V86 memory and adjusts the pointer accordingly.

The final macro terminates the interpretation of the script and reflects an Int 21h into the V86 portion of the VM. When the Int 21h returns, both buffers will be freed.

You can combine any of the macros, although you should keep in mind that **Xlat_API_Exec_Int** and **Xlat_API_Jmp_To_Proc** both terminate interpretation of the current script.

WARNING You should always specify the exact length of a buffer or else strange things may occur. For example, it is incorrect to translate an API that has a maximum buffer size of 128 bytes by using the Xlat_API_Fixed_Len macro if the buffer can be smaller than 128 bytes. This can cause bugs if the program has data that is updated at interrupt time that is located past the end of the buffer.

For example, assume a program has the following data:

```
Buffer_Length   db      64
Buffer_Data     db      64 dup (?)
```

```
Time_Of_Day     dd      0
Other_Stuff     db      500 dup (?)
```

Assume the program updates the Time_Of_Day field from the timer interrupt. If the translation code copies 128 bytes of data starting with Buffer_Length into V86 mode memory and while processing the call a timer interrupt executes then the Time_Of_Day field will be incremented. However, when the buffer is copied back the old time will be copied on top of the current (correct) Time_Of_Day field.

V86MMGR_Load_Client_Ptr

Description

This service will load **FS:ESI** with the specified Client_Seg:Offset. If the VM is running a 16-bit protected mode then the high word of the offset in **ESI** will be zeroed. Otherwise, if the VM is running a 32-bit program or is in **VxD_Exec_Mode** then the high word of **ESI** will not be zeroed. This allows most translation procedures to operate correctly without the need to test the execution mode of the current VM.

The value passed in **AX** should be formed from the Client Register Structure equates. For example, to load the VM's **DS:(E)DX** you would use the following code:

```
mov     ax, (Client_DS * 100h) + Client_DX
VxDcall V86MMGR_Load_Client_Ptr
(FS:ESI -> Same address as Client_DS:(E)DX).
```

Entry

VM must be in protected mode
AH = Client segment register equate
AL = Client offset register equate
EBX = Current VM Handle
EBP -> Client register structure

Exit

FS:ESI -> Client's buffer

Uses

FS, ESI, Flags

V86MMGR_Allocate_Buffer

Description

Allocates a portion of the current VM's translation buffer and optionally copies data from the PM pointer in **FS:ESI** into the allocated buffer.

Note that this service will map fewer bytes than the value specified in the ECX parameter if the length of the buffer extends past the FS segment limit. Therefore, you need to preserve the value returned in ECX from this service to use when deallocating the buffer using **V86MMGR_Free_Buffer**.

The buffers are maintained as a stack. Therefore, the last buffer allocated must be the first buffer freed.

Entry

Current VM must be in protected mode
EBX = Current VM Handle
EBP -> Client register structure
ECX = Number of bytes to allocate
FS:ESI = Pointer to extended memory to copy
If carry flag is set then
    Source buffer will be copied into V86 buffer
else
    Source buffer will not be copied into V86 memory

Exit

If carry set then
    ERROR: Could not allocate buffer (out of space)
else
    ECX = Actual number of bytes allocated (<= original ECX)
    High WORD of **EDI** = V86 segment of translation buffer
    Low WORD of **EDI** = Offset of allocated buffer

Uses

ECX, **EDI**, Flags

V86MMGR_Free_Buffer

Description

Deallocates a buffer that was allocated by the **V86MMGR_Allocate_Buffer** service. It will optionally copy data from the translation buffer to the buffer pointed to by **FS:ESI**.

The buffers are maintained as a stack. Therefore, the last buffer allocated must be the first buffer freed.

Entry

Current VM must be in protected mode
EBX = Current VM Handle
EBP -> Client register structure
ECX = Number of bytes to free (returned from Allocate_Buffer)
FS:ESI = Pointer to extended memory buffer
If carry flag is set then
    Buffer will be copied from V86 memory before buffer freed
else
    Buffer will not be copied

Exit

None

Uses

Flags

V86MMGR_Get_Xlat_Buff_State

Description This service returns information about the current mapping buffer status.

> ***WARNING*** Always call this service to find the segment of the translation buffer. Since the buffer can move at any time you should never make any assumptions about the size or location of the buffer.

Entry **EBX** = VM handle (any VM handle valid)

Exit **EAX** = V86 segment of translation buffer (high word 0)
ECX = Number of bytes of buffer not in use
EDX = Total size of buffer in bytes (max size 10000h)

Uses **EAX, EBX, ECX,** Flags

V86MMGR_Set_Xlat_Buff_State

Description This service is used to switch to an alternate mapping buffer. This feature is provided for protected mode terminated-and-stay resident programs which may need to switch to a private translation buffer before executing protected mode DOS calls since the default buffer may be full.

You should get the current translation buffer state, set the new state, perform any DOS call, and then set the state back to the original values.

Entry **EBX** = VM handle (any VM handle valid)
EAX = V86 segment of translation buffer (high word 0)
ECX = Number of bytes of buffer not in use
EDX = Total size of buffer in bytes (max size 10000h)

Exit None

Uses Flags

V86MMGR_Get_VM_Flat_Sel

Description This service returns a selector that points to the base of the specified VM's V86 address space. This is useful for 32-bit applications since this selector can be used to point to any address in the VM's V86 address space. The selector is writeable and has a limit of 11,000h bytes so that the high memory area is also addressable.

The selector returned is in the specified VM's LDT. Therefore, the selector is only valid to use when the VM is running (is the current VM).

Entry EBX = VM handle (any VM handle is valid)

Exit EAX = Selector with base at high linear addr of V86 memory (high word 0)

Uses EAX, Flags

V86MMGR_Get_Mapping_Info

Description This service will return information about the current page mapping areas.

Entry None

Exit CH = Number of pages reserved for global mapping (total)
CL = Number of pages available (not in use) for global mapping

V86MMGR_Map_Pages

Description This service maps the specified buffer into every VM at the same address using page mapping. If the contents of memory are changed in one VM, the change will be reflected in the original buffer as well in all other VMs.

Entry ESI -> Linear address to map
ECX = Number of bytes to map

Exit

```
If carry flag is set then
    ERROR: Could not map memory
else
    Memory is mapped
    ESI = Map handle (used to free the map region)
    EDI = Linear address of map buffer (< 1 meg)
```

Uses ESI, EDI, Flags

V86MMGR_Free_Page_Map_Region

Description This service will "unmap" pages that were mapped by the **V86MMGR_Map_Pages** service.

Entry ESI = Map handle to free

Exit Old map buffer address contains null memory
ESI is undefined

Uses ESI, Flags

Chapter 41 Virtual DMA Device (VDMAD) Services

The VDMAD virtualizes DMA (Direct Memory Access) I/O for standard DMA channels for all VMs. By default, it handles all programmed I/O for the DMA controllers and arbitrates I/O to the physical DMA ports so that more than one VM can be using the same DMA channels at the same time. In some cases, the default handling of DMA channels is not desirable. To handle these cases, VDMAD provides a number of services to enable another VxD to take control of the virtualization of specific DMA channels.

VDMAD also provides some services that can be used by Bus Master devices that have their own DMA controllers. These devices still need to be able to lock and unlock DMA regions in memory and determine the physical addresses of these regions. Bus Master devices can also make use of the buffer services, if they cannot otherwise scatter/gather a linear region that is not physically contiguous.

The VDMAD services available for Bus Master use are as follows:

- **VDMAD_Copy_From_Buffer**
- **VDMAD_Copy_To_Buffer**
- **VDMAD_Default_Handler**
- **VDMAD_Disable_Translation**
- **VDMAD_Enable_Translation**
- **VDMAD_Get_EISA_Adr_Mode**
- **VDMAD_Get_Region_Info**
- **VDMAD_Get_Version**
- **VDMAD_Get_Virt_State**
- **VDMAD_Lock_DMA_Region**
- **VDMAD_Mask_Channel**
- **VDMAD_Release_Buffer**
- **VDMAD_Request_Buffer**
- **VDMAD_Reserve_Buffer_Space**
- **VDMAD_Scatter_Lock**

- **VDMAD_Scatter_Unlock**
- **VDMAD_Set_EISA_Adr_Mode**
- **VDMAD_Set_Phys_State**
- **VDMAD_Set_Region_Info**
- **VDMAD_Set_Virt_State**
- **VDMAD_Unlock_DMA_Region**
- **VDMAD_UnMask_Channel**
- **VDMAD_Virtualize_Channel**

VDMAD_Copy_From_Buffer

Description This service allows another device to copy data from the VDMAD buffer to the actual DMA region associated with the buffer. This service is called after **VDMAD_Request_Buffer**, after a memory write transfer and before **VDMAD_Release_Buffer**.

Entry
EBX = buffer ID
ESI = region linear
EDI = offset within buffer for start of copy
ECX = size

Exit
Carry clear
data copied from buffer into DMA region
Carry set
AL = 0Ah (DMA_Invalid_Buffer) - invalid buffer id supplied
= 0Bh (DMA_Copy_Out_Range) - (ESI + ECX) is greater than buffer size

Uses Flags

VDMAD_Copy_To_Buffer

Description This service allows another device to copy data into the VDMAD buffer from the actual DMA region associated with the buffer.This service is called after **VDMAD_Request_Buffer** and before starting a memory read transfer.

Entry
EBX = buffer id
ESI = region linear

EDI = offset within buffer for start of copy
ECX = size

Exit

Carry clear
data copied from DMA region into buffer
Carry set
AL = 0Ah (DMA_Invalid_Buffer) - invalid buffer id supplied
= 0Bh (DMA_Copy_Out_Range) - (**ESI** + **ECX**) is greater than buffer size

Uses

Flags

VDMAD_Default_Handler

Description

Default DMA channel I/O callback routine. This routine receives notifications of virtual state changes and handles setting up the physical state to start DMA transfers.

```
get virtual state
If channel virtually unmasked then
    lock region
    If lock fails then
       request buffer
       If memory read opeartion then
           copy data to buffer
    set phyical state
    physically unmask channel
```

Entry

EAX = DMA handle
EBX = VM handle

Exit

None

Uses

Anything

VDMAD_Disable_Translation

Description

This service disables the automatic translation done for the standard DMA channels.It is necessary, if a V86 app or driver, or a PM app uses the DMA services thru INT 4BH to determine actual physical addresses for DMA transfers. A disable count is maintained, so a matching call to **VDMAD_Enable_Translation** is required for each call to this service to re-enable translation.

Entry	EAX = DMA handle EBX = VM Handle
Exit	Carry clear automatic translation is disable for the channel Carry set the disable count overflowed
Uses	Flags

VDMAD_Enable_Translation

Description	This decrements the disable count associated with a standard DMA channel. If the disable count goes to 0, then automatic translation is re-enabled. See **VDMAD_Disable_Translation** for further information.
Entry	EAX = DMA handle EBX = VM Handle
Exit	Carry clear service completed successfully Z-flag clear, if automatic translation is re-enabled Carry set attempt to enable when translation already enabled
Uses	Flags

VDMAD_Get_EISA_Adr_Mode

Description	Get EISA extended mode - the hardware doesn't allow for reading the extended mode for a channel, so VDMAD defaults to the ISA defaults (channels 0-3 are byte channels and 5-7 are word channels with word addresses and counts) An INI switch can specify an alternate setting.
Entry	EAX = Channel # (0..7) or DMA Handle
Exit	CL = 0 - 8-bit I/O, with count in bytes CL = 1 - 16-bit I/O, with count in words and adr shifted CL = 2 - 32-bit I/O, with count in bytes CL = 3 - 16-bit I/O, with count in bytes

Uses ECX, Flags

VDMAD_Get_Region_Info

Description Get information about the current region assigned to a DMA handle. This information can be used by a handler to call the following services:

- **VDMAD_Unlock_DMA_Region**
- **VDMAD_Release_Buffer**
- **VDMAD_Copy_To_Buffer**
- **VDMAD_Copy_From_Buffer**

Entry EAX = DMA handle

Exit
BL = buffer id
BH = pages locked (0 = FALSE, else TRUE)
ESI = region linear
ECX = size in bytes

Uses EBX, ECX, ESI

VDMAD_Get_Version

Description Returns the version of the Virtual DMA Device

Entry None

Exit
AH = Major version number
AL = Minor version number
ECX = Buffer size in bytes (0, if not allocated; a buffer will always be allocated, but it doesn't happen until **Device_Init**)
Carry flag clear

Uses EAX, Flags

VDMAD_Get_Virt_State

Description

This service allows a channel owner to determine the current virtual state of the channel. The virtual state consists of all the information necessary to physically program the DMA channel for a DMA transfer (linear address of target region, byte length of region, mode of transfer, and state of mask bit and software request bit) This state information reflects how the VM thinks the hardware is currently programmed.

Entry

EAX = DMA handle
EBX = VM handle

Exit

If translation is enabled
  ESI = high linear address of the user's DMA region
      (high linear is used so that the DMA can proceed
      even if a different VM is actually running at the
      time of the transfer)
Else
  ESI = physical byte address programmed (shifted left 1,
      for word ports)
ECX = count in bytes
DL= mode (same as 8042 mode byte with channel # removed
      and DMA_masked & DMA_requested set as
      appropriate:
          DMA_masked   channel masked and not ready
                          for a transfer
          DMA_requested   software request flag set)
DH= extended mode (ignored on non-PS2 machines that don't
          have extended DMA capabilities)

Uses

ESI, ECX, EDX, flags

VDMAD_Lock_DMA_Region

Description

This service attempts to lock a region of memory for a DMA transfer. It is called before a DMA transfer is started (before the physical state is set for a channel and before it is unmasked.)

It first verifies that the region is mapped to contiguous pages of physical memory.

Then it determines whether the region will result in a DMA bank (page)

wrap

On AT class machines each channel has a base address register and a page address register. The base address register is incremented after each byte or word transfered. If the increment of this 16 bit register results in the roll over from FFFFh to 0, then the transfer wraps to the start of the DMA bank because the page register is not updated. Normally DOS watches for this condition and adjusts INT 13h parameters to split transfers to avoid this wrap, but DOS doesn't know anything about the difference between linear and physical addresses under enhanced Windows, so VDMAD checks again to prevent wrap from occurring undesirably.

If all of these checks are okay, then the service calls the memory manager to lock the physical pages.

NOTE This routine does not check to see if the region is within some physical maximum constraint. If the region is lockable, then it locks the memory, and it is up to the caller to check to see if the physical region is acceptable. If the region is not acceptable, then the caller should unlock the region and perform a buffered DMA transfer.

Entry

ESI = linear address of actual DMA region
ECX = # of bytes in DMA region
DL = 1b, if region must be aligned on 64K page boundary
= 10b, if region must be aligned on 128K page boundary

Exit

Carry set, if lock failed
ECX = # of bytes that are lockable in the region
(starting from ESI)
AL = 1 (DMA_Not_Contiguous), region not contiguous
= 2 (DMA_Not_Aligned), region crossed physical
alignment boundary
= 3 (DMA_Lock_Failed), unable to lock pages
ELSE
EDX = physical address of the DMA region
the region has been locked

Uses

EAX, ECX, EDX, Flags

VDMAD_Mask_Channel

Description

This service physically masks a channel so that it will not attempt any further DMA transfers.

Entry

EAX = DMA handle

Exit

None

Uses

Flags

VDMAD_Release_Buffer

Description Release the VDMAD buffer assigned to a DMA channel from a previous **VDMAD_Request_Buffer** call. This routine exits from a critical section and the DMA buffer will now be available for other users. Any data in the buffer is not automatically copied, so **VDMAD_Copy_From_Buffer** must be called if the data is important.

Entry **EBX** = Buffer ID

Exit
Carry clear
  buffer released
Carry set
  bad ID

Uses Flags

VDMAD_Request_Buffer

Description This service reserves the DMA buffer for a DMA transfer.

Entry
ESI = linear address of actual DMA region
ECX = # of bytes in DMA region

Exit
Carry clear
  EBX = buffer ID
  EDX = the physical address of the buffer
Carry set
  AL = 5 (DMA_Buffer_Too_Small), region request is too large for buffer
    = 6 (DMA_Buffer_In_Use), buffer already in use

Uses **EAX, EBX, ESI,** Flags

VDMAD_Reserve_Buffer_Space

Description This service allows other devices that are going to handle DMA to make sure that VDMAD allocates a buffer large enough for any transfers that they might require. It also allows a device to specify a maximum physical address that would be valid for the device's DMA requests (such as 1Mb for an XT.) During the **Device_Init** phase of initialization, VDMAD will allocate the DMA buffer using all of the contraints specified by other devices.i.e. the buffer will be at least as big as the largest size specified by the calls to this service, and it will be allocate below the lowest maximum physical addresses specified.

This service is only available during Sys_Critical_Init.

Entry

EAX = # of pages requested
ECX = maximum physical address that can be included in a
DMA transfer; 0, if no limit.

Exit

None

Uses

Flags

VDMAD_Scatter_Lock

Description

This service attempts to lock all pages mapped to a DMA region and return the actual physical addresses of the pages.

Entry

EBX = VM Handle
AL = 0, if the DDS table should be filled with physical
addresses and sizes of the physical regions that
make up the DMA region
AL = 1, if the DDS table should be filled with the actual
page table entries
AL = 3, if the DDS table should be filled with the actual
page table entries and not present pages should not
be locked
EDI -> extended DDS (DMA Descriptor Structure)

Exit

Carry clear
Z-flag set
whole region was locked successfully
Z-flag clear
partial region locked
Carry set
nothing locked

EDX = # of table entries needed to describe whole region
DDS_size = # of bytes locked
DDS table has been updated
if request was for page table copy (AL=1 OR 3), then
ESI = offset into first page for start of the region

Uses

EDX, ESI, Flags

VDMAD_Scatter_Unlock

Description This service attempts to unlock all pages locked by a previous call to VDMAD_Scatter_Lock

Entry

EBX = VM Handle
AL = 0, if the DDS table should be filled with physical addresses and sizes of the physical regions that make up the DMA region
AL = 1, if the DDS table should be filled with the actual page table entries
AL = 3, if the DDS table should be filled with the actual page table entries and not present pages should not be locked
EDI -> extended DDS (DMA Descriptor Structure)
(The table at the end of the DDS is not required, so it is not necessary to maintain the table for this unlock call.)

Exit

Carry clear
Lock counts have been decremented. If no other VxD's had pages locked, then the pages have been unlocked.
Carry set
The memory was not locked.

Uses Flags

VDMAD_Set_EISA_Adr_Mode

Description Set EISA extended mode

Entry

EAX = Channel # (0..7) or DMA Handle
CL = 0 - 8-bit I/O, with count in bytes
CL = 1 - 16-bit I/O, with count in words and adr shifted
CL = 2 - 32-bit I/O, with count in bytes
CL = 3 - 16-bit I/O, with count in bytes

Exit None

Uses Flags

VDMAD_Set_Phys_State

Description This service programs the DMA controller state for a channel. All that it needs to know is the desired mode. The location and size of the buffer is taken from the information passed to the service **VDMAD_Set_Region_Info** which must be called previously.

Entry
EAX = DMA handle
EBX = VM handle
DL = mode
DH = extended mode

Exit None

Uses Flags

VDMAD_Set_Region_Info

Description Set information about the current region assigned to a DMA handle. This service must be called before calling **VDMAD_Set_Phys_State.**

Entry
EAX = DMA handle
BL = buffer id
BH = pages locked (0 = FALSE, else TRUE)
ESI = region linear
ECX = size in bytes
EDX = physical address for transfer

Exit None

Uses Flags

VDMAD_Set_Virt_State

Description Modify the virtual state of a DMA channel. This is service is used when a channel owner wants to change the virtual state of a channel from how the VM programmed it.This might be used to split a DMA request into smaller pieces, etc.

Entry
EAX = DMA handle
EBX = VM handle
If translation is enabled
  ESI = high linear address of the user's DMA region

(high linear is used so that the DMA can proceed
even if a different VM is actually running at the
time of the transfer)
Else
ESI = physical byte address programmed (shifted left 1,
for word ports)
ECX = count in bytes
DL= mode (same as 8042 mode byte with channel # removed
and DMA_masked & DMA_requested set as
appropriate:

DMA_masked	channel masked and not ready for a transfer
DMA_requested	software request flag set)

DH= extended mode (ignored on non-PS2 machines that don't
have extended DMA capabilities)

Exit None

Uses Flags

VDMAD_Unlock_DMA_Region

Description This service unlocks the DMA region previously locked to a channel. It is called after a DMA transfer is complete and the channel has been masked. So that the controller will not attempt any further transfers to the programmed address.

Entry **ESI** = linear address of actual DMA region
ECX = # of bytes in DMA region

Exit Carry clear
memory unlocked
Carry set
error

Uses Flags

VDMAD_UnMask_Channel

Description This service physically unmasks a channel so that DMA transfers can proceed.

Entry **EAX** = DMA handle
EBX = VM Handle

Exit None

Uses Flags

VDMAD_Virtualize_Channel

Description This service allows another VxD to claim ownership of a standard DMA channel. The new owner registers a callback routine that will be called whenever the virtual state of the channel is changed as a result of I/O done in a VM. In some cases a device doesn't want to allow a VM to perform DMA to a channel at all (they will handle programming based on a private API, etc. instead of virtualized hardware I/O), so it is possible to pass a 0 to specify a null callback routine. VDMAD will continue to trap the I/O for the channel, but won't ever change the physical state of the channel as a result of any VM I/O.

Entry
EAX is Channel #
ESI is I/O Callback procedure (0 = none)

Exit
Carry set if channel is already owned
ELSE
  EAX is DMA handle

Uses Flags

Callback
ENTRY
  EAX = DMA handle
  EBX = VM handle
  Proc can modify **EAX, EBX, ECX, EDX, ESI, EDI**, and flags

EXIT
  None

Appendixes

Appendix A Terms and Acronyms

The following list explains the terms and acronyms that are found in the Device Development Kit for Windows 3.0.

B

Banding The process of dividing a display surface such as a page into smaller rectangles, composing those individual bands within memory, and then sending the output to the printer one band at a time.

C

Clipping The process of removing any portion of a graphic image that extends beyond a specified boundary.

Control Block A per Virtual Machine (VM) data structure in which Virtual Devices (VxDs) and the Virtual Machine Manager (VMM) can maintain the VM's state information.

Control Panel A Windows application that lets you change system settings, including printer assignments and characteristics.

D

Device Driver The dynamic-link library that provides the hardware-dependent, low-level interface between Windows GDI functions and the graphics output device.

Dynamic Data Exchange (DDE) A protocol that cooperating programs can use to exchange data without user intervention.

Dynamic-Link Library (DLL) A library with which an application is fixed up upon initial loading. **(needs improving)**

Device Independent Bitmap (DIB) A bitmap format that can be interpreted and converted by a device driver into its own specific format. It is called "device independent" because any driver capable of using DIBs can display (or otherwise use) the DIB to the best of its ability.

E

Escape A device-dependent operation that is not supported by the device-independent GDI module. The entry point in the device driver is called **Control()**; in GDI (i.e., to the application), it is called **Escape()**.

F

Font Resource A group of individual fonts that have various combinations of heights, widths, and pitches.

G

Graphics Device Interface (GDI) A device-independent, high-level graphics manager. GDI provides the interface that feeds graphics commands from Windows application programs to the device driver.

GDI Library A set of supporting functions for device drivers. These utilities include versions of output functions such as **Bitblt** and **Strblt**, a Transpose function for banding devices, and priority queue functions for daisywheel printers.

I

IOPM I/O Permission Map

M

Metafile A collection of GDI function calls stored in a binary coded form and used to transfer device-independent pictures between programs.

Microsoft Macro Assembler (MASM) An assembly language compiler. Version 5.0 includes increased speed (25% faster than 4.0), simplified segment declarations, support for the 80386 and 80387 processors, a version of CodeView that's compatible with four languages, utilities to aid in program development, and completely revised manuals.

N

Non-Windows Application A program that does not make use of the Windows environment. Instead, it calls MS-DOS and the BIOS, and accesses the hardware directly.

P

Paging A capability used by enhanced Windows by which any linear address (defined by segment: offset) in the system can be mapped to any physical memory.

Palette The range of colors that the video adapter can display and manage.

Pixel The smallest element of a physical display surface that can be independently assigned color or intensity.

Pixel Array A matrix of pixels that defines the color for a region on an actual display. There is exactly one pixel definition for each addressable picture element of a raster display covered by the pixel array.

Presentation Level Protocol (PLP) A standard protocol used for transmitting high quality text.

Primitive A basic graphic function to be performed.

Print Manager The Windows utility that prints files without suspending the operation of other programs. It also enables you to change the priority of print jobs or to cancel them.

Printer Command Language (PCL) The language used by Hewlett-Packard ® Laserjet ® and compatible printers.

Protected Mode (PM) A mode of the 80386 processor that provides a linear address of 4 gigabytes per segment and 16K segments, thereby breaking the 640K barrier and giving applications access to much more memory. Windows and Windows applications run in protected mode. VxD's must handle access from both protected mode and virtual 8086 mode.

R

Raster Device A device that uses a matrix of pixels covering the entire screen or page area (display or printed surface) to draw graphics. Pixels (points) are turned on and off, bit-by-bit.

Resolution The number of visibly distinct dots that can be displayed in a given area of the screen. Typical resolution is 100 dots per inch.

Red, Green, Blue (RGB) Values from a color table. This color table is used in mapping from a color index to corresponding color values.

S

Scaling Coordinate scaling transforms points from one level to another. GDI scales coordinates from NDC space to values appropriate for your graphics device.

System VM The first Virtual Machine (VM) under enhanced Windows. The VM in which Windows runs.

T

TSRs Terminate-and-Stay Resident applications

V

Vector Device A device that draws graphics with lines. Beginning and ending points are set and a line is drawn between them.

Virtual Device Interface (VDI) The ANSI graphics interface upon which GDI is based. VDI is a standard interface between device-dependent and device-independent code in a graphics environment. VDI makes all device drivers appear identical to the application program.

VDMAD Virtual DMA Device

VDD Virtual Display Device

VKD Virtual Keyboard Device

VMD Virtual Mouse Device

VPICD Virtual Programmable Interrupt Controller Device

Virtual 8086 mode (V86) A mode of the 80386 processor by which the 80386 emulates the function of the 8086 processor. In this mode, each segment has a linear address limit of 64K and the applications can address a total of 1M + 64K - 16 bytes.

Virtual "x" Device (VxD) The name of the device virtualized replaces the "x" in this name. There must be a VxD for each piece of hardware that can have a different state in each of the VMs. Any piece of hardware that does not have an associated VxD is global. It must handle interleaved access from multiple VMs or have a global piece of software (such as a DOS device driver or TSR) that serializes access to the hardware. All the VxDs run in the same, flat-model, 32-bit segment as the rest of the VMM. A VxD can also provide services that are not directly associated with a piece of hardware (e.g., a piece of code that replaces an MS-DOS or BIOS service).

Virtual Machine (VM) The collective state of an instance (maintained in the control block) of the VMM and the VxDs, and the memory associated with the program executing in the VM. This includes all the code and data in virtual 8086 mode as well as protected mode.

Virtual Machine Manager (VMM) The core of enhanced Windows. It runs, along with all the VxDs, in one, flat-model, 32-bit segment.

W

WDEB386 An enhanced Windows version 3.0 debugger program.

Window A rectangular region on a display screen in which the system displays the contents of an application.

Windows Application Any program that has been specifically designed to run under Microsoft Windows.

WIN.INI The Windows initialization file in which you maintain the system-wide settings. This is a text-based file that resides under the Windows software directory.

Appendix B
Understanding Modes

Windows 3.0 documentation uses the term "mode" in overlapping ways. This appendix is provided to clarify the different uses.

B.1 Windows Modes

To provide the greatest features for the available hardware, Windows 3.0 can run in three software modes: real, standard, or 386 enhanced. The following table compares the memory models and required microprocessor for each of these Windows modes.

Windows 3.0	Real Mode	Standard Mode	386 Enhanced Mode
Supported Memory Model	Real Mode	Real Mode Protected Mode (16-bit)	Real Mode Protected Mode (32-bit) V86 Mode
Required Hardware	8086 80286 80386 80486	80286 80386 80486	80386 80486

B.2 Microprocessor Modes

As the Intel microprocessors evolved greater capabilities, they continued to support the programs and operating systems of the earlier architectures. As a result, the 80386 has no fewer than four modes. Each is compared below to the earlier architectures.

The first is the familiar *real-mode*, wherein the 80386 functions as a fast 8086/88-compatible processor with some bonus opcodes. Like the 80286, the 80386 always powers up in real mode and can, therefore, run any existing 8086 operating systems and software.

In *protected-mode*, the 80386 can take on two different personalities. It can execute a logical superset of the 80286 protected-mode instructions and run 16-bit programs. Or, while in its native protected mode, it can use 32-bit instructions, registers, and stacks and can allow individual memory segments as large as 4GB. The native protected mode also has an additional level of address translation—supported in hardware by page tables—that allows much greater flexibility in mapping the linear address onto physical memory. In either protected mode, the 80386 translates selectors and offsets to linear addresses using descriptor tables in much the same manner as the 80286.

The forth operating mode, *virtual 86 mode* (V86), provides another form of 8086 emulation. But now, instead of a single program running in a single memory partition, the 80386 can create multiple partitions, each capable of running a real-mode program. Each partition has its own address space, I/O port space, and interrupt vector table. Enhanced Windows uses the V86-mode partitions to create virtual machines, the fundamental components in its virtual machine architecture. The architecture is described in Chapter 16, "Overview of Windows in 386 Enhanced Mode."

The following table summarizes the four modes of the 80386 microprocessor:

Mode	Description
Real Mode	Functions as a very fast 8086/88-compatible processor.
Protected Mode (16-bit)	Functions in protected mode as an enhanced 286 processor.
Protected Mode (32-bit, native mode)	Functions in protected mode using full 32-bit instructions, registers, and stacks.
Virtual 86 Mode	Runs multiple, protected, virtual 8086 machines, each with its own 1MB of memory space.

Appendix C
Creating Distribution Disks for Drivers

Not available for this release.

Appendix D
Windows INT 2FH API

Enhanced Windows 3.0 supports an Application Program Interface (API) designed to enable DOS device drivers, TSR programs, and application programs to take full advantage of the multitasking abilities of the enhanced Windows environment.

Most application program writers will use the interface that releases the current virtual machine's time-slice. This API allows enhanced Windows and OS/2 to multitask non-Windows specific DOS applications more efficiently. The Release Time Slice API should be used by applications even if they are not running under enhanced Windows. This allows OS/2 to detect idleness in DOS applications. OS/2 will recognize the enhanced Windows release time-slice call but it does not support other enhanced Windows APIs.

The Microsoft 80286 DOS extender will issue the initialization and exit INT 2FH API calls so that real mode software can free extended memory through XMS. The 286 DOS extender also supports the Int 31h service detection Int 2FH API call.

Other APIs are used by DOS device drivers and TSRs that have enhanced Windows specific requirements.

D.1 Call-In Interfaces

Call-in interfaces are APIs that real mode DOS device drivers, TSRs, and applications use to communicate with enhanced Windows. These include:

- Get Windows version
- Get virtual machine ID
- Begin critical section
- End critical section
- Release time slice
- Get device API entry point
- Switch VMs and callback

D.1.1 Enhanced Windows Installation Check (AX=1600H)

This API call is valid under all versions of enhanced Windows. If a program intends to use a enhanced Windows API, it must first make sure that the enhanced Windows environment is running. To do this issue:

```
mov     ax, 1600h
int     2Fh
test    al, 7Fh
jz      Not_Running_Win386
(Otherwise enhanced Windows is running)
cmp     al, 1
je      Running_Ver_2xx
cmp     al, -1
je      Running_Ver_2xx
(Else al contains major version, AH contains minor)
```

If 0 or 80H is returned in AL, enhanced Windows is not running. Any other value means that enhanced Windows is running. A value of 1 or -1 (0FFH) indicates that the application is running under enhanced Windows version 2.0 or 3.0. Otherwise, AL will contain the major version number (3 or higher) and AH will contain the minor version number. The table below summarizes the possible return values:

Value in AL	Meaning	
00H	Enhanced Windows 3.x or Windows/386 version 2.xx is not running	
80H	Enhanced Windows 3.x or Windows/386 version 2.xx is not running	
01H	Windows/386 version 2.xx running	
FFH	Windows/386 version 2.xx running	
Anything else	AL = Major version number	AH = Minor

D.1.2 Releasing Current Virtual Machine's Time-Slice (AX=1680h)

NOTE This API should be used only by non-Windows specific applications. Windows programs should yield their time by calling the *WaitMessage* function.

This API is used by programs to indicate that the program is idle (usually waiting for the user to type something). By issuing this interrupt, applications prevent enhanced Windows from wasting time running a program that is essentially doing nothing. This allows other programs to use the time.

Programs should always use this API even if they are not Windows-specific applications and even if they are not currently running under Windows in 386 enhanced mode. This allows OS/2 to detect idleness even though it does not support the complete enhanced Windows API. The only check you should make before issuing the API call is to make sure that the INT 2FH interrupt vector is not zero.

Sample code:

```
        mov     ax, 352Fh
        int     21h             ; DOS get vector 2Fh
        mov     ax, es          ; ES:BX = Vector
        or      ax, bx          ; Q: Is it zero?
        jz      Skip_Idle_Call  ;    Y: Skip this
        mov     ax, 1680h       ;    N: Tell Win
        int     2Fh             ;       we're idle.
Skip_Idle_Call:
```

If the API is supported, the INT 2FH will return with AL=0, otherwise it will return with AL unchanged (80h). Usually application programs will not be interested in the return value.

Note that when an application uses this API it will continue to run occasionally so your program should re-issue the interrupt in the program's idle loop. In other words, this API does NOT block your application until a key is pressed.

D.1.3 Begin Critical Section (AX=1681h)

If a DOS device driver or TSR needs to prevent a task-switch from occurring, it should call this interface. When a virtual machine is in a critical section, no other task will be allowed to run except to service hardware interrupts. For this reason, the critical section should be freed (using the end critical section API) as soon as possible.

D.1.4 End Critical Section (AX=1682h)

This API must be called to release ownership of the critical section that was claimed using the Begin Critical Section API. Every call to Begin Critical Section must be followed by a matching call to End Critical Section.

D.1.5 Get Current Virtual Machine ID (AX=1683h)

This API returns with BX = Current virtual machine ID. The ID is unique for each virtual machine. Although Windows currently runs in VM 1, your software should not rely on this. Also, if a VM is destroyed, its ID may be reused by another new virtual machine. Be sure to treat VM IDs as a word (not a byte). An ID of 0 will *never* be returned.

D.1.6 Get Device API Entry Point (AX=1684h)

Some VxDs (enhanced Windows device drivers) provide a set of services that application programs can access. For example, the Virtual Display Device provides services that the Windows old application program uses to display DOS programs in a window. Any VxD can support an API for DOS applications. Your program must issue an INT 2FH with AX=1684h and BX = Virtual device ID. The entry point address will be returned in ES:DI. Your application must execute a FAR CALL to this address to call the virtual device. If the value returned is 0:0 then the device does not support an API, otherwise ES:DI is the address of the procedure to call. You should either make sure your application is running on version 3.0 or zero **ES** and **DI** before using this API.

```
xor     di, di   ; * Only necessary if you have   *
mov     es, di   ; * not checked for Win ver 3.0 *
mov     ax, 1684h
mov     bx, My_Device_ID
int     2Fh
mov     ax, es
or      ax, di
jz      API_Is_Not_Supported
(else API address in ES:DI)
```

The definition of a device API is specified by the virtual device driver. Refer to individual virtual device documentation for details.

D.1.7 Switch VMs and CallBack (AX=1685h)

Some DOS devices, such as networks, need to perform functions in a specific virtual machine. These devices can use this interface to force the appropriate virtual machine to be installed so that they can modify the VM's data. Refer to Chapter 24, "Primary Scheduler Services," for information on appropriate priority boosts.

Entry: **AX** = 1685h
BX = VM ID of virtual machine to switch to
CX = Flags
Bit 0 = 1 if wait until interrupts enabled
Bit 1 = 1 if wait until critical section unowned
All other bits must be 0
DX:SI = Priority boost (**DX**=High word, **SI**=Low word)
ES:DI = **CS:IP** of procedure to call

Exit:

```
If carry set then
   AX = Error code
else
   Event will be called or has been called already.
```

Error codes: 1 = Invalid VM ID
2 = Invalid priority boost
3 = Invalid flags

Callback procedure: Must save all registers modified
Must IRET to caller
Priority will remain boosted until procedure irets

D.1.8 Detect Presence of INT 31H Services (AX=1686h)

If a program needs to detect the presence of the INT 31H protected mode API, it can use this INT 2FH. Note that this particular API is also supported by the Microsoft 80286 DOS extender for protected mode Windows. INT 31H services are only supported for protected mode programs.

Entry

AX = 1686h

Exit

```
If AX = 0 then
    INT 31H services are available and can be called
else (AX != 0)
   INT 31H services are not available
```

D.2 Call Out Interfaces

Enhanced Windows will broadcast INT 2FH to real mode device drivers and TSRs to inform them of various activities. These can be used to load enhanced Windows installable devices, free extended memory, instance per-VM data structures, and turn on or off various device services or features. For example, SmartDrv can free extended memory for enhanced Windows to use when the initialization call is made and then reclaim it when it receives the termination call. DOS devices such as networks can inform the enhanced Windows loader to load a special protected mode installable device that cooperates with the real mode network device driver.

D.2.1 Enhanced Windows and 286 DOS Extender Initialization (AX=1605h)

The enhanced Windows loader and the Microsoft 286 DOS extender will broadcast an INT 2FH with the following parameters:

AX = 1605h
ES:BX = 0:0
DS:SI = 0:0
CX = 0
DX = Flags
Bit 0 = 0 if enhanced Windows initialization
1 if Microsoft 286 DOS extender initialization
All other bits reserved and undefined.

Any DOS device driver or TSR can hook Int 2FH and watch for this particular broadcast. When this broadcast is received, the real mode software can inform enhanced Windows or the 286 DOS extender that it should not load by returning with **CX** != 0. The TSR or device that fails the initialization should print an error message so the user can take appropriate steps to reconfigure the machine. Enhanced Windows and the Microsoft DOS extender will not print an error message—they will only issue the termination API call and return to DOS.

If it is OK for enhanced Windows or the DOS extender to load, the real mode software should not modify CX and may want to do one or more of the following:

- Release extended memory through the XMS interface.
- Switch back to real mode (if currently in virtual 8086 mode) or set **DS:SI** to the Virtual 8086 mode enable/disable routine address.
- Load an installable device (enhanced Windows only).
- Instance private data structures (enhanced Windows only).

The DOS extender only pays attention to the value returned in CX. It will not instance any data or load enhanced Windows installable device drivers. The DOS extender only issues this call so that extended memory can be released and the machine can be placed in real mode if it is currently in virtual 8086 mode.

Instance data refers to data in a TSR or DOS Device driver that must have a private copy in each VM. Normally, all TSRs and devices loaded before enhanced Windows is run are considered global memory. That means that all of the data is shared between virtual machines. However, there are some pieces of data that actually should be maintained on a per-VM basis. For example, the DOS command line buffer needs to be instanced (this is done automatically by enhanced Windows). However, TSRs such as the DOS command line editors will not function properly unless they identify the data that needs to be instanced.

The first two options (release extended memory, or switch from V86 to real mode) are up to the device to handle on its own. The last options require returning a pointer to a list of structures to load. Your INT 2FH hook must first chain to the next INT 2FH handler with all registers unmodified. When the handler returns you must take the **ES:BX** value returned and place it in the following data structure in the *Next_Dev_Ptr* field:

```
Win386_Startup_Info_Struc STRUC
SIS_Version                 db      3, 0
SIS_Next_Dev_Ptr            dd      ?
SIS_Virt_Dev_File_Ptr       dd      0
SIS_Reference_Data          dd      ?
SIS_Instance_Data_Ptr       dd      0
Win386_Startup_Info_Struc ENDS
```

Your software must point **ES:BX** at this structure and return. This allows multiple enhanced Windows installable devices to be loaded through a single INT 2FH call.

The **SIS_Version** field is used by enhanced Windows to determine the size of the structure. This field should always contain 3, 0 to indicate that it is version 3.0.

The **SIS_Next_Dev_Ptr** points to the next structure in the list. This field must be filled in with the returned **ES:BX** after your software chains to the next INT 2FH handler.

SIS_Virt_Dev_File_Ptr is a pointer to an ASCIIZ string that contains the name of a enhanced Windows virtual device file. DOS devices such as networks use this to force a special enhanced Windows protected mode virtual device to be loaded. If this field is zero, then no device will be loaded.

The **SIS_Reference_Data** is only used when the **SIS_Virt_Dev_File_Ptr** is non-zero. This DWORD will be passed to the virtual device when it is initialized. The DWORD can contain any value. Often it contains a pointer to some device specific data structure.

The **SIS_Instance_Data_Ptr** field points to a list of data to be instanced. If the field is zero, then no data will be instanced. Each entry in the list has the following structure:

```
Instance_Item_Struc STRUC
        IIS_Ptr         dd      ?
        IIS_Size        dw      ?
Instance_Item_Struc ENDS
```

The list is terminated with a zero DWORD.

Your handler must preserve all registers except the values returned in **ES, BX,** and **CX**. It must also preserve **DS** and **SI** unless it explicitly changes them to return the address of the virtual 8086 mode enable/disable routine. Remember, any device that returns with **CX** != 0 will force enhanced Windows or the 286 DOS extender to abort. If the load is aborted, the termination INT 2FH will be issued immediately.

Enhanced Windows supports loading with a virtual mode program such as an EMM "LIMulator" running provided that the program supports a virtual 8086 mode enable/disable callback routine. The address of the routine must be returned in **DS:SI**. If your TSR or device driver sets this return parameter, it should first check to make sure that **DS** and **SI** are both zero. If they are non-zero, then fail the initialization by setting CX=non-zero. Notice that the Microsoft 286 DOS extender will not call this routine. Therefore, you must either set the processor into real mode during the initialization INT 2FH or set CX=non-zero to abort the load.

The virtual mode enable/disable callback will be called with **AX=0** to disable V86 mode (return to real mode) and **AX=1** to re-enable V86 mode. Just before attempting to enter protected mode enhanced Windows will disable V86 mode after every VxD has been loaded. It will call the enable/disable routine with **AX=0** and with interrupts disabled. Do not enable interrupts in your routine unless the routine will return with Carry set to indicate failure. After enhanced Windows exits, it will call the enable/disable routine in real mode with **AX=1** and with interrupts disabled to set the machine back into V86 mode.

The enable/disable routine will be called with a FAR return frame. It must return with the carry flag clear to indicate success or Carry set to indicate an error. If an error is returned from the disable call, then enhanced Windows will abort. The error return from the enable V86 call will be ignored and the machine will be left in real mode. It is the responsibility of the enable/disable routine to print an error message.

D.2.2 Enhanced Windows and 286 DOS Extender Exit (AX=1606h)

When enhanced Windows or the 286 DOS extender terminates it will broadcast an INT 2FH with the following parameters:

AX = 1606H.
DX = Flags
Bit 0 = 0 if enhanced Windows exit
1 if Microsoft 286 DOS extender exit
All other bits reserved and undefined.

This call will be issued in real mode. It allows devices and TSRs to undo anything they did when enhanced Windows or the DOS extender initialized. For example, a device like SmartDrv may re-allocate extended memory that it released during initialization.

If the initialization broadcast fails (returns with CX != 0) then this broadcast will be issued immediately.

D.2.3 Device Call Out API (AX=1607h)

This API is, in reality, more of a convention than an API. It specifies a standard mechanism for enhanced Windows virtual devices to talk to DOS device drivers and TSRs.

Some devices need to ask real-mode DOS software for information. For example, the Virtual NetBIOS mapper VxD will issue an INT 2FH to determine if a network supports an extended NetBIOS API. The standard device call out will have AX=1607H and BX=Device ID. As with the device API entry point call-in interface, the details of the interface are specified by the device that issues the interrupt.

This interrupt may be issued at any time, either in real mode or after enhanced Windows has begun execution.

D.2.4 Enhanced Windows Initialization Complete (AX=1608h)

This API call is made by enhanced Windows after all the installable devices have initialized. At this point, it is still possible to identify instance data and perform other functions that are restricted to enhanced Windows initialization time. The enhanced Windows device initialization phase is complete, so it is possible to call enhanced Windows device API entry points.

WARNING Real mode software such as a TSR or DOS device driver may be called after the enhanced Windows initialization call and before this API call is made. It is the responsibility of the real mode software to detect and properly handle this situation.

D.2.5 Enhanced Windows Begin Exit (AX=1609H)

This API call is issued at the beginning of a normal Enhanced Windows exit sequence. It is sent at the start of the **Sys_VM_Terminate** device control call phase. All VxDs still exist so calls to device API entry points are still valid.

WARNING This call will not be made in the event of a fatal system crash. Also, real mode code may be executed after this API call has been made and before enhanced Windows has returned to real mode. It is the responsibility of the real mode software to detect and properly handle these situations.

D.3 Windows/386 Version 2.xx API Compatibility

The release of Windows/386 (version 2.xx) had a limited Application Program Interface that was defined to help support real mode DOS device drivers such as networks. The 2.xx API allows DOS programs to:

- Determine if Windows/386 or enhanced Windows (version 3.0) is running
- Get the ID of the current Virtual Machine
- Enter and leave a global critical section

The APIs used under version 2.xx are fairly complex and inflexible. We suggest that, unless your application or device driver absolutely needs to run under version 2.xx, you ignore all version 2.xx APIs and use the 3.0 APIs instead.

D.3.1 Installation Check

To test for Windows/386 version 2.xx you should issue an Int 2fh with **AX=1600h**. Refer to Windows Installation Check for complete documentation for this API call.

D.3.2 Determining the Current Virtual Machine (Get VM ID)

Once the software has determined that it is running under Windows/386 version 2.xx it must make another call to get the API procedure address. To do this issue:

```
mov     ax, 1602h
int     2fh
(ES:DI -> Windows/386 API procedure)
```

The API procedure is the same address for every virtual machine, so you will need to issue this call only once (although you can issue it as often as you want).

To get the ID of the current virtual machine *jump* to the Windows/386 API procedure with **AX** = 0 and **ES:DI** - the address to return to.

Sample code:

```
        mov     di, cs
        mov     es, di
        mov     di, OFFSET Win386_AIP_Return
        xor     ax, ax                  ; AX = 0
        jmp     [Win386_API_Proc]
Win386_API_Return:
        (Now BX contains the current VM ID)
```

Note that you must place the return address in **ES:DI** and JUMP to the API procedure. When Win386 returns control to your program it will return to **ES:DI.**

This interface is supported under version 3.0 only for compatibility reasons. New DOS devices or applications should use the version 3.0 interface.

D.3.3 Critical Section Handling

Windows/386 version 2.xx does not support API calls to enter and leave a critical section. However, by incrementing and decrementing a special DOS critical section counter called the InDOS flag, you can force the current virtual machine into a critical section. Incrementing InDOS is not sufficient to enter a critical section in version 3.0. You will need to make an API call first and then, if it fails, increment the InDOS flag.

To get the address of the InDOS flag issue the following DOS call (documented in *The MS-DOS Encyclopedia*):

```
        mov     ah, 34h
        int     21h
        (ES:BX -> InDOS flag)
```

The InDOS flag is a byte within the MS-DOS kernel. The value in InDOS is incremented when MS-DOS begins execution of an Interrupt 21H and decremented when MS-DOS's processing of that function has completed. When you increment the byte, current versions of enhanced Windows will not switch to another virtual machine. Therefore, to enter a critical section, you need to increment the byte and to leave a critical section you should decrement the InDOS flag.

WARNING You must make sure your code never decrements the InDOS flag through zero. DOS will set the InDOS flag to zero under some error conditions (for example, the user types CTRL+C). Also, even if the InDOS flag is non-zero, an Int 28H may cause the VM to be rescheduled.

For versions 3.xx and greater of Windows you will need to issue an INT 2FH **AX** = 1681H to begin a critical section and **AX** = 1682H to end a critical section. Note that if a program enters the critical section N times, it must also issue the end critical section interrupt N times before the critical section is actually released. Thus, nested begin/end critical section calls are valid. Both of these APIs will zero the **AL** register to indicate that the critical section API is supported. You should *not* increment and decrement InDOS under versions of Windows that support these API calls.

Unlike the InDOS critical section method, an INT 28H will *not* reschedule the current virtual machine. The only way a task switch will occur is by completely releasing the critical section.

Since you need to call the Windows API or increment the InDOS flag you will probably want to write two procedures similar to the following:

```
Begin_Win_Critical_Section:
        push    ax
        mov     ax, 1681h
        int     2Fh
        test    al, al
        jz      BCS_Quick_Exit
        push    es
        les     ax, [InDOS_Address]
        inc     BYTE PTR es:[ax]
        pop     es
BCS_Quick_Exit:
        pop     ax
        ret

End_Win_Critical_Section:
        push    ax
        mov     ax, 1682h
        int     2Fh
        test    al, al
        jz      ECS_Quick_Exit
        push    es
        les     ax, [InDOS_Address]
        cmp     BYTE PTR es:[ax], 0
        je      (Error handler routine)
        dec     BYTE PTR es:[ax]
        pop     es
ECS_Quick_Exit:
        pop     ax
        ret
```

Index

G

H

I

L

N

P

R

S