Using VToolsD™

Release 2.7

Windows® 95
Windows® 98
Windows® Me

Part Number 11555

June, 2002

Restricted Rights Legend

Software License Agreement

This Software License Agreement is not applicable if Licensee has a valid Compuware License Agreement and has licensed this Software under a Compuware Product Schedule.

COMPUWARE CORPORATION ("COMPUWARE") IS WILLING TO LICENSE THIS SOFTWARE (SOFTWARE) TO YOU ONLY ON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS AGREEMENT. BEFORE YOU CLICK ON THE "ACCEPT" BUTTON, CAREFULLY READ THE TERMS AND CONDITIONS OF THIS AGREEMENT. BY CLICKING ON THE "ACCEPT" BUTTON, YOU ARE CONSENTING TO BE BOUND BY THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE TO ALL OF THE TERMS AND CONDITIONS OF THIS AGREEMENT, THEN COMPUWARE IS UNWILLING TO LICENSE THE SOFTWARE TO YOU, IN WHICH EVENT YOU SHOULD CLICK THE "DO NOT ACCEPT" BUTTON TO DISCONTINUE THE INSTALL PROCESS AND CONTACT YOUR COMPUWARE SALES REPRESENTATIVE.

Software License Agreement

Please Read This License Carefully

You, either an individual or entity (Licensee), are purchasing a license (Agreement) to use this Compuware Corporation software, related user manuals (Documentation), and other materials provided hereunder (collectively, the Software). The Software is the property of Compuware Corporation (Compuware) and/or its licensors, is protected by intellectual property laws, and is provided to Licensee only under the license terms set forth below. This Agreement does not transfer title to the intellectual property contained in the Software. Compuware reserves all rights not expressly granted herein. By clicking on the "Accept" button and/or installing the Software indicates your acceptance of these terms. If you do not agree to all of the terms and conditions of this Agreement, do not install the Software and contact a Compuware sales representative.

Title and Proprietary Rights: Licensee acknowledges and agrees that the Software is proprietary to Compuware and/or its licensors, and is protected under the laws of the United States and other countries. Licensee further acknowledges and agrees that all rights, title and interest in and to the Software, including intellectual property rights, are and shall remain with Compuware and/or its licensors. Unauthorized reproduction or distribution is subject to civil and criminal penalties.

Evaluation Copy: This section shall apply **only** if the Software has been provided for Licensee's evaluation of the Software (Evaluation Copy). An Evaluation Copy is provided AS IS, with no warranties, express or implied, or maintenance service; for the sole and exclusive purpose of enabling Licensee to evaluate the Software. The Evaluation Copy will automatically time-out at the end of the evaluation period. If Licensee elects to continue to use the Software at the end of the evaluation period, Licensee must contact a Compuware representative.

Use Of The Software: Compuware grants Licensee the limited right to use the Software included in the package with this license, subject to the terms and conditions of this Agreement. Licensee agrees that the Software will be used solely for internal purposes. Licensee may not use the Software to offer data processing services to third parties, including but not limited to timesharing, facilities management, outsourcing or service bureau use, or any other third party commercial purpose or gain. Only one copy of the Software may be installed on a single computer at any one time unless:

(i) The Software is designed and intended by Compuware for use in a shared network client server environment, as set forth in the Documentation; and

(ii) Licensee agrees to provide technical or procedural methods to prevent use of the Software, even at different times, by anyone other than Licensee; and

(iii) Licensee has purchased a license for each individual user of the Software and/or for each computer that will have access to the Software. Any unauthorized use of this Software may cause termination of this Agreement.

Licensee may make one machine-readable copy of the Software for BACK UP PURPOSES ONLY. This copy shall display all proprietary notices, be labeled externally to show that it is the property of Compuware, and that its use is subject to this Agreement. Documentation may not be copied in whole or part. Licensee agrees to provide technical or procedural methods to prevent use of the Software by anyone other than Licensee, even at different times. Licensee may not use, transfer, assign, export or in any way permit the Software to be used outside the country of purchase, unless authorized in writing by Compuware. Except as expressly provided in this Agreement, Licensee may not modify, reverse engineer, decompile, disassemble, distribute, sublicense, sell, rent, lease, give or in any way transfer the Software, by any means or in any medium, including telecommunications. Licensee will use its best efforts and take all reasonable steps to protect the Software from unauthorized use, copying or dissemination, and will retain all proprietary notices intact.

Maintenance Service: Licensee may subscribe to maintenance service on an annual basis by paying the maintenance fee then in effect. If Compuware provides maintenance service while connected to Licensee's system or network either remotely or in person, Compuware shall not be responsible for any damage or loss, including but not limited to server failure, data loss, virus/worm infection and network downtime. Licensee is responsible for maintaining adequate backup and virus intrusion software for its server, data and network systems.

Redistribution Rights of Device Driver Development Software: This section shall **only** apply if the Software is device driver development software, used by Licensee to develop application or device driver programs (User Software), as specified in the Documentation. The User Software may include run-time components (RTC's) that have been extracted by the Software from the library files of the Software, programs to remotely test the User Soft-

ware, and compiled code examples. These RTCs, examples, and programs are specifically designated as redistributable in the Documentation. Licensee has a non-exclusive, royalty-free, restricted license to:

(i) Modify, compile, and distribute the driver code examples;
(ii) Distribute the remote testing program for testing purposes only;
(iii) Embed the RTCs and driver code examples in its User Software, in object code form only; and
(iv) Reproduce and distribute the RTCs and driver code examples embedded in its User Software, in object code form only, provided that:
(a) Licensee distributes the RTCs and driver code examples only in conjunction with and as a part of its User Software;
(b) Licensee will be solely responsible to anyone receiving its User Software for any updates, technical and other support obligations, and any other liability which may arise from the distribution of its User Software;
(c) Licensee does not use Compuware's or its licensors' names, logos, or trademarks to market or distribute its User Software;
(d) Licensee includes Compuware's and its licensors' copyright and/or proprietary notices and legends within the executable images of its User Software and on Licensee's software media and documentation; and
(e) Licensee agrees to indemnify, hold harmless and defend Compuware and its licensors from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of its User Software.

Government Users: With respect to any acquisition of the Software by or for any unit or agency of the United States Government, the Software shall be classified as "commercial computer software," as that term is defined in the applicable provisions of the Federal Acquisition Regulation (the "FAR") and supplements thereto, including the Department of Defense (DoD) FAR Supplement (the "DFARS"). If the Software is supplied for use by DoD, the Software is delivered subject to the terms of this Agreement and either (i) in accordance with DFARS 227.7202-1(a) and 227.7202-3(a), or (ii) with restricted rights in accordance with DFARS 252.227-7013(c)(1)(ii) (OCT 1988), as applicable. If the Software is supplied for use by a Federal agency other than DoD, the Software is restricted computer software delivered subject to the terms of this Agreement and (i) FAR 12.212(a); (ii) FAR 52.227-19; or (iii) FAR 52.227-14(ALT III), as applicable. Licensor: Compuware Corporation, 31440 Northwestern Highway, Farmington Hills, Michigan 48334.

Limited Warranty and Remedy: Compuware warrants the Software media to be free of defects in workmanship for a period of ninety (90) days from purchase. During this period, Compuware will replace at no cost any such media returned to Compuware, postage prepaid. This service is Compuware's sole liability under this warranty. COMPUWARE DISCLAIMS ALL EXPRESS AND IMPLIED WARRANTIES, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO LICENSEE. IN THAT EVENT, ANY IMPLIED WARRANTIES ARE LIMITED IN DURATION TO THIRTY (30) DAYS FROM THE DELIVERY OF THE SOFTWARE. LICENSEE MAY HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.

Infringement of Intellectual Property Rights: In the event of an intellectual property right claim, Compuware agrees to indemnify and hold Licensee harmless, provided Licensee gives Compuware prompt written notice of such claim, permits Compuware to defend or settle the claim, and provides all reasonable assistance to Compuware in defending or settling the claim. In the defense or settlement of such claim, Compuware may obtain for Licensee the right to continue using the Software or replace or modify the Software so that it avoids such claim, or if such remedies are not reasonably available, accept the return of the infringing Software and provide Licensee with a pro-rata refund of the license fees paid for such Software based on a three (3) year use period.

Limitation of Liability: LICENSEE ASSUMES THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE. IN NO EVENT WILL COMPUWARE BE LIABLE TO LICENSEE OR TO ANY THIRD PARTY FOR ANY SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING BUT NOT LIMITED TO, LOSS OF USE, DATA, REVENUES OR PROFITS, ARISING OUT OF OR IN CONNECTION WITH THIS AGREEMENT OR THE USE, OPERATION OR PERFORMANCE OF THE SOFTWARE, WHETHER SUCH LIABILITY ARISES FROM ANY CLAIM BASED UPON CONTRACT, WARRANTY, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR OTHERWISE, AND WHETHER OR NOT COMPUWARE OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO LICENSEE. IN NO EVENT SHALL COMPUWARE BE LIABLE TO LICENSEE FOR AMOUNTS IN EXCESS OF PURCHASE PRICE PAID FOR THE SOFTWARE.

Term and Termination: This License Agreement shall be effective upon Licensee's acceptance of this Agreement and shall continue until terminated by mutual consent, or by election of either Licensee or Compuware in case of the other's unremediated material breach. In case of any termination of this Agreement, Licensee will immediately return to Compuware the Software that Licensee has obtained under this Agreement and will certify in writing that all copies of the Software have been returned or erased from the memory of its computer or made non-readable.

General: This Agreement, the Compuware invoice and Licensee purchase order, are the complete and exclusive statement of the parties' agreement and supersede any prior agreement or understanding whether oral or written, relating to the subject of this Agreement. The preprinted terms of any purchase order accepted by Compuware are expressly superseded by this Agreement. Should any provision of this Agreement be held to be invalid by any court of competent jurisdiction, that provision will be enforced to the maximum extent permissible and the remainder of the Agreement shall nonetheless remain in full force and effect. This Agreement shall be governed by the laws of the State of Michigan and the United States of America.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS.

Table of Contents

Preface

Congratulations on selecting NuMega VTOOLSD™ for your Virtual Device Driver (VxD) development project. Whether you are an experienced VxD developer or you are about to implement your first device, VTOOLSD can save you a significant amount of time compared to the more traditional assembly language approach.

Windows VxDs have traditionally been the domain of the assembly language programmer. Until now, the lack of appropriate tools for building VxDs in high level languages, coupled with a need for high performance and small code size, has made assembly language the only realistic choice for most VxD programmers.

While assembly language still has a role in VxDs, there is a growing need for tools that facilitate writing VxDs in high level languages, such as C and C++. C was invented as a systems programming language, and operating systems such as UNIX and Windows NT are written primarily in C. Now you can build VxDs in C to combine the power and expressiveness of a high level language with the efficiency and fine control of assembly language.

The use of C++ in systems programming is a relatively recent development. In the case of VxDs, it is an innovation introduced by VTOOLSD. The VTOOLSD class library provides a structure for VxDs that is analogous to an "application framework" for applications in event driven environments. By taking an object-oriented approach, the developer can design the VxD based on high level abstractions, rather than in terms of the hundreds of assorted services that the operating system interface makes available. The class library provides a natural model for operating system objects that simplifies VxD programming and increases productivity.

VTOOLSD provides a comprehensive approach to VxD development in C or in C++, assisting from the initial prototype to the retail build. Whether you choose C or C++, VTOOLSD will cut your development time and costs significantly. Here are some of the highlights:

VTOOLSD gets you started fast.	QuickVxD is an interactive application that gives you a fast start on a new VxD project. According to parameters you specify, it generates a make file, an include file, and function skeletons for a new VxD in either C or C++.
VTOOLSD libraries are comprehensive.	The VTOOLSD libraries include C and C++ interfaces for every service offered by the Virtual Machine Manager and all of the standard Microsoft VxDs.
VTOOLSD libraries are designed for VxD development.	All of the VTOOLSD libraries are specifically designed for use in VxDs. The libraries are structured to ensure that all code is linked into the proper segment.
VTOOLSD libraries are designed for performance.	Large portions of the libraries are written in assembly language to ensure optimal performance and small size. Each routine is packaged in a separate object module to conserve memory; you never need to link extraneous code.
VTOOLSD includes an ANSI C run time library.	VTOOLSD includes a broad subset of the ANSI C run-time functions rewritten expressly for VxD development.
VTOOLSD does not sacrifice performance.	VTOOLSD documents how to easily combine assembly language modules with VxDs written in C or C++. Or, use the in-line assembler within your C or C++ code.
VTOOLSD supports development in C.	The VTOOLSD C Framework is a set of functions and macros that simplify coding the infrastructure of your VxD. It defines data structures and performs basic services in a clean, modifiable, and efficient manner.
VTOOLSD supports development in C++.	The VTOOLSD class library provides dozens of classes that comprise an object-oriented interface for VxD programming. Use the class library to structure your VxD, to handle a variety of events, and to build your own reusable software components.
VTOOLSD automates the VxD build process.	The VTOOLSD make system is driven by a small set of parameters that you control. The make system relieves you from having to figure out the correct compiler switches, include directories, segment attributes, library searches, symbol file generation commands, and all the other details that could slow down your development effort.
VTOOLSD includes debugging support.	Each of the libraries is provided in Debug and Retail versions. Debug versions of the libraries contain diagnostic code that can detect various error conditions. VTOOLSD includes macros and services you can use in debug builds to track down bugs quickly.

VTOOLSD includes full on-line help.	VTOOLSD includes on-line reference files documenting the VTOOLSD libraries, all VMM and VxD services, control messages, and important data structures.
VTOOLSD includes library source code.	Source code for all of the VTOOLSD libraries is included with the package as a reference.
VTOOLSD supports novice and experienced VxD programmers.	The manual and on-line documentation include valuable background material on VxD programming and principles. Examples illustrate key concepts in complete, ready-to-run VxDs.
VTOOLSD is supported.	Compuware provides technical support for all aspects of VTOOLSD. Technical support is available through a variety of means, including telephone, fax, and on-line services.

Windows 3.1

Windows 95

The manual describes VxD development for Windows 3.1, Windows 95, and Windows 98. Information specific to only one version is noted throughout the text with the [Windows 3.1] and [Windows 95] symbols illustrated on the left.

What is Redistributable?

The following files, included with VTOOLSD, are redistributable.

- Monitor.exe
- Monitor.hlp
- Monitor.cnt
- Ndbgmsg.vxd
- All files installed to the LIB folder (redistributable when built into drivers)
- All files installed in the EXAMPLES folder (redistributable when built into drivers, not as source files)

1 Product Description

VTOOLSD is a programmer's toolkit designed to simplify the development of Virtual Device Drivers (VxDs) for Microsoft Windows. VTOOLSD includes libraries for C and C++ programming, extensive on-line documentation, and introduces the QuickVxD program for fast development of VxD skeletons.

VTOOLSD offers complete support for C and C++ programming. Wrap Libraries allow direct access to almost all VMM and standard VxD services from C/C++ code. The extensive Class Library provides powerful constructs for C++ programmers who wish to build VxDs using an object-oriented approach. The C Run-Time Library offers a rich set of ANSI-compatible C run-time functions designed for use in virtual devices.

This chapter briefly describes the components of the VTOOLSD toolkit.

Programs and Utilities

The following utility programs are installed in the BIN subdirectory of the VTOOLSD installation.

QuickVxD

QuickVxD is a Windows application designed to give you a fast start on a new VxD. QuickVxD generates customized source code that serves as the basis for your device driver.

Monitor

Monitor enables you to load and unload VxDs from the desktop to view debug information emitted by VxDs.

VxDView

VXDVIEW lets you examine information about all the VxDs currently loaded on your system. It automatically updates its information whenever a VxD is loaded or unloaded.

Segalias

SEGALIAS is a utility program that operates on OMF object files, such as those produced Borland C++ version 4.x. SEGALIAS alters the segment names generated by the compiler so they conform to VxD conventions. SEGALIAS also preprocesses object files of VxDs built with the PELE utility.

PELE

PELE converts 32-bit DLLs from PE (Portable Executable) format to LE (Linear Executable) format, the latter being the format that VxD files employ. PELE enables building of VxDs with any linker that can produce a standard Windows 32-bit executable.

Sethdr

SETHDR is a utility program that replaces ADDHDR, the Microsoft utility. Its primary function is to store extra driver information in the header of the VxD file. SETHDR can also display information from the VxD header, set the device name, append a resource file, and detect discrepancies between the header and the Device Data Block.

VxDVer

VxDVer creates a compiled version resource that SETHDR appends to a VxD.

Libraries

The VTOOLSD libraries are designed to simplify the task of VxD programming in C or C++. Each library is distributed in both retail and debug versions. The VTOOLSD libraries are installed in the LIB subdirectory.

Detailed documentation for each of the VTOOLSD libraries is provided in the on-line help files.

Wraps

The wrap libraries provide a C interface to all of the Virtual Machine Manager and standard VxD services, callable from either C or C++ programs. These libraries support all of the services provided by the Virtual Machine Manager and the standard Microsoft VxDs. The file INCLUDE\VXDSVC.H contains the prototypes for all the wrapper functions.

The wrap libraries also include thunk code that allows all callbacks and interrupt handlers to be written in C.

C Framework

The CFrame libraries provide a framework for writing VxDs in C without requiring any assembly language.

Class Library

The class libraries provide an object-oriented framework for VxD development in C++. These extensive libraries offer class-based access to most Virtual Machine Manager and VxD services and provide new features through a set of additional classes designed to simplify common VxD programming tasks.

C Run-Time Library

The C Run-time libraries provide a rich set of ANSI C library functions. These functions, written specifically for use in the VxD environment, can be called from either C or C++.

NDIS Libraries

VTOOLSD includes object libraries for implementing drivers that call the Network Device Interface Specification (NDIS) functions.

Device Access Architecture (DAA) Library and NT Emulation Library

The DAA library, supported by the NT emulation library, is a class library compatible with DriverWorks tool kit. See the online help for the set of classes that are available.

Library Source Code

Compuware provides source code for all of the VTOOLSD libraries (excluding the NDIS wrapper libraries). The assembler for the corresponding compiler is required. All source code resides in subdirectory SOURCE. Due to the large number of modules, some of the source code is installed in .ZIP format. In order to obtain the best possible compression, some library source files have been zipped twice. Simply use an unzip utility to expand the "flat" packed file, then unzip again to retrieve the individual source files.

Make Files

VTOOLSD provides make files that knit together the many tools required to build VxDs. The VTOOLSD make files work with the make files you construct for your project. The make system is described in detail in Chapter 5.

The VTOOLSD make files are installed in the INCLUDE subdirectory with the .MAK extension.

Header Files

The header files distributed with VTOOLSD are used in place of the header files provided in the DDK and with the C/C++ compiler. These files are described in detail in the chapters on VTOOLSD Basics and in the C Framework and Class Library chapters.

All of the header files are installed in the INCLUDE subdirectory.

Online Help

VTOOLSD includes online help files that document the library functions, control messages, data structure, procedures, and utility programs in the VTOOLSD toolkit.

The help files are installed in the BIN subdirectory.

Examples

VTOOLSD includes a number of complete VxD examples written in C and C++. The C examples are installed in subdirectories of the EXAMPLES\C subdirectory; the C++ examples are installed in subdirectories of the EXAMPLES\CPP subdirectory. New examples are added to the kit regularly; see the online help for a description of the examples in the currently shipping version.

Technical Support

For Non-Technical Issues

Customer Service is available to answer any questions you might have regarding upgrades, serial numbers and other order fulfillment needs. Customer Service is available from 8:30am to 5:30pm EST, Monday through Friday. Call:

- In the U.S. and Canada: 1-888-283-9896
- International: +1-603-578-8103

For Technical Issues

Technical Support can assist you with all your technical problems, from installation to troubleshooting.

Before contacting technical support please read the relevant sections of the product documentation and the ReadMe files.

You can contact Technical Support by:

E-Mail	Include your serial number and send as many details as possible to mailto:nashua.support@compuware.com
World Wide Web	Submit issues and access additional support services at: **http://frontline.compuware.com/nashua/**
Fax	Include your serial number and send as many details as possible to 1-603-578-8401
Telephone	Telephone support is available as a paid* Priority Support Service from 8:30am to 5:30pm EST, Monday through Friday. Have product version and serial number ready. In the U.S. and Canada, call: 1-888-686-3427 International customers, call: +1-603-578-8100 * Technical Support handles installation and setup issues free of charge.

Before contacting Technical Support, please obtain and record the following information:

- Product/service pack name and version.
- Product serial number.
- Your system configuration: operating system, network configuration, amount of RAM, environment variables, and paths.
- The details of the problem; settings, error messages, stack dumps, and the contents of any diagnostic windows.
- If the problem is repeatable, the details of how to create the problem.
- The name and version of your compiler and linker and the options you used in compiling and linking.

2 Understanding VxDs

Microsoft Windows is a complex operating system. Its job, like that of any operating system, is to manage and abstract the resources of the machine. The clients of the operating system are applications, including both Windows and DOS programs.

The resources managed by the operating system include memory, processor time, and access to specific hardware such as the display, keyboard, mouse, serial ports, and any other devices that are attached to the system.

Ideally, an operating system provides perfect protection and isolation between applications. This is accomplished by providing a "virtual machine" for each application. In the ideal model, each application runs as if it were the only application running on the system. The operating system provides a separate address space for each application and manages contention for resources when more than one application attempts to allocate and use a device.

Like most real-world products, the true situation is somewhat less than ideal. The layered architecture of the Windows operating system, combined with limitations of the real-mode DOS substrate, provide some of the benefits of full virtualization, but lack the protection of full isolation that complete virtualization would imply.

Windows is a Layered Operating System

The Windows operating system is implemented in two distinct layers. The upper layer provides the API services that are familiar to Windows application developers, including all of the services provided by USER, GDI, and KERNEL. These services provide a convenient foundation for building end-user applications. This layer manages the resource known as the *display*, allowing multiple applications to share the screen. In addition, the upper layer provides memory management to Windows applications.

All Windows applications, along with the Windows API services that constitute the upper layer of the operating system, run in a single *virtual machine* known as the System VM. Each DOS session is started in a new virtual machine; there can be many virtual machines active simultaneously.

The Virtual Machine Manager

The lower layer of the operating system, known as the *Virtual Machine Manager*, is hidden from most application writers. The Virtual Machine Manager creates and manages multiple virtual machines.

The Virtual Machine Manager *virtualizes* the hardware resources of the machine to give each virtual machine the appearance that it has exclusive control of the system. The Virtual Machine Manager also provides virtual memory and includes a scheduler that allocates processor time between the various virtual machines.

The Virtual Machine Manager consists of a core set of services, plus numerous modules that are loaded when Windows starts. These additional modules are known as Virtual Device Drivers, or *VxDs.* Some VxDs are provided by Microsoft and support devices that must be present, such as the keyboard; display, and even low level components of the computer such as the Direct Memory Access (DMA) controller and Programmable Interrupt Controller (PIC). Other VxDs are developed independently and are installed to support specific hardware or software.

Because VxDs are loaded as part of the lowest layer of the operating system, they operate with more privileges than other applications. None of the normal limits, restrictions, or protection mechanisms apply to Virtual Devices.

Developers have taken advantage of the power available to VxDs, originally intended to virtualize hardware devices, for a wide variety of purposes.

What This Chapter Covers

This chapter explains how VxDs work, describes the basic VxD programming model, and discusses some of the services VxDs can call.

Developers who are not familiar with the Virtual Machine Manager/VxD environment should read this chapter carefully. Experienced VxD writers might choose to skim this material or skip it entirely.

What You Should Already Know

You should have a basic understanding of the architecture of the Intel x86 family of microprocessors in order to understand the material presented in this chapter. In particular, it will be helpful if you are familiar with at least the general concepts of:

- Protected and virtual 8086 mode operation

- Flat memory model
- Interrupt and exception processing
- Protection and privilege levels
- Segment and page memory management and fault handling
- Input and output protection and fault handling

For more information on the Intel processor family architecture, refer to Intel's *Programmer's Reference Manual* for the 80386 or later processors, or one of the many third party books that cover similar material.

The Virtual Machine Manager Architecture

Before introducing the specifics of VxD construction, let us discuss the architecture of the Virtual Machine Manager in more detail.

The Virtual Machine Manager is a 32-bit protected mode operating system. Its primary responsibility is to create, run, monitor, and terminate virtual machines. The Virtual Machine Manager provides services that manage memory, tasks, interrupts, and protection faults. Initially, the Virtual Machine Manager controls and manages all hardware devices in the system.

VxDs are 32-bit protected mode dynamic-link libraries that extend the Windows operating system. Under the supervision of the Virtual Machine Manager, VxDs cooperatively manage system hardware and respond to a wide range of software and hardware events.

Processor Privilege Level

The 80x86 architecture has many features that aid operating systems in maintaining system integrity. One such important feature is privilege levels, or *rings*, which are used to isolate applications from sensitive system code and data and other applications. The privilege level at which any given code is running determines what memory regions that code can access and which instructions it can execute. The processor raises a fault condition if the privilege rules are violated.

Windows uses two of the available privilege levels. The Virtual Machine Manager runs in ring 0, the most privileged protection level. VxDs also run in ring 0. All applications and DLLs run at less privileged protection level 3.

Description of Windows Virtual Machines

A *virtual machine* is a task that has its own address space, I/O port space, interrupt vector table, and CPU registers.

Every virtual machine has a virtual 8086 (V86) component, an optional protected mode (PM) component, and a set of ring-0 data structures maintained by the VMM.

V86 Component

The V86 component of a virtual machine contains a one megabyte address space. Windows uses the Virtual 8086 mode of the processor to run DOS applications in a virtual machine as if they were running in real mode. When a new virtual machine is created, the Virtual Machine Manager initializes the virtual machine's V86 address space with a copy of every program that was running in real mode when Windows was started. This includes DOS as well as DOS Device Drivers and TSRs that are already resident.

All Windows applications are protected mode applications, but not all protected mode applications are necessarily Windows applications. References to protected mode applications indicate either Windows or DOS-extended applications. References to DOS, V86, or real mode applications refer to Virtual 8086 programs.

PM Component

Programs running in a virtual machine can use the DOS Protected Mode Interface (DPMI) to enter protected mode. The Windows 16-bit ring 3 executive (KRNL386.EXE) uses DPMI services to enter protected mode. All Windows applications are protected mode applications, that is, they are loaded into extended memory and run in protected mode.

Ring-0 Component

The Virtual Machine Manager maintains a set of data structures for each virtual machine. This data includes a ring-0 stack that is used when faults or interrupts cause control to transfer from an application to the operating system. The VMM also maintains scheduling information and instance data (see below) for each VM.

When control transfers from the V86 or PM component of a virtual machine into the ring-0 component, the VM's CPU registers are saved in a structure called the *Client Register Struct* located on its ring-0 stack.

It is important to understand that the system is always running in the context of a particular virtual machine, even when executing privileged ring-0 VMM or VxD code.

Windows Memory Map

Windows 95

Windows 95 partitions the processor's 4GB linear address space into four regions called *arenas*. From bottom to top, the four arenas are: DOS, Private, Shared, and System.

Arena	Occupied by	Lower Bound	Upper Bound
INVALID	Illegal pointers	MAXSYSTEMLADDR+1 (top - 4 Meg = 0xffc00000)	TOP OF MEMORY (4 Gig = 0xffffffff)
SYSTEM	VMM and VxDs	MINSYSTEMLADDR (3 Gig = 0xbfffffff)	MAXSYSTEMLADDR (top - 4 Meg = 0xffbfffff)

SHARED	16-bit Windows apps and DLLs; 32-bit DLLs; DPMI memory	MINSHAREDLADDR (2 Gig = 0x80000000)	MAXSHAREDLADDR (3 Gig = 0xbfffffff)
PRIVATE	Win32 apps (context-switched)	MINPRIVATELADDR (4 Meg = 0x400000)	MAXPRIVATELADDR (2 Gig = 0x7fffffff)
DOS	V86-mode apps, DOS device drivers, TSRs	MINDOSLADDR (0)	MAXDOSLADDR (4 Meg = 0x003fffff)

Windows 95

The Private arena pertains only to the System VM. In this arena, the system maps storage for the private code and data of Win32 applications. The Windows 95 scheduler time-slices Win32 applications. Each time an application receives the execution focus, the pages of that application's memory context are mapped into the private arena. A Win32 application cannot address the private pages of another Win32 application.

The highest 4 MB of the address space are not accessible by applications or VxDs. This provides a means to create an intentionally illegal pointer.

DOS Arena Memory Map

Within the DOS arena, the address is partitioned into distinct regions. The following table describes these regions, starting with the highest in memory, and going down to linear address zero.

Region Available	Purpose
VM CB Heap	Control block heap is used to allocate private storage for VxDs that maintain per-VM data.
HMA	High memory area is used as auxiliary storage for DOS tables or resident code.
ROM	Contains the system BIOS.
UMB	The upper memory blocks can contain a portion of the DOS heap or resident code.
Video Memory	Used by the display.
V86 Private Area	DOS applications started under Windows use this region. To obtain the bounds of this region, call GetFirstV86Page and GetLastV86Page.
V86 Global Area	MS-DOS, device drivers, TSR's. This memory is common to all VM's except for pages that have been instanced using AddInstanceItem.

In order to save memory, the Virtual Machine Manager uses the paging mechanisms of the processor so that only a single copy of shared code and data needs to be maintained. Thus, only a single copy of much of DOS and BIOS is stored in memory.

Pages that contain data specific to a single virtual machine are *instanced*. This means that separate copies, or *instances*, of the page are maintained.

This technique provides a means for virtualizing data in the VM address space. Applications can only access the current instance of the data; other instances do not appear in the VM's address space at all.

Managing Virtual Machine

The System Virtual Machine

The Virtual Machine Manager creates the system virtual machine early during system initialization. Windows is loaded into the context of the system virtual machine.

Creation

A new virtual machine is created whenever a new non-Windows application is started from Windows. The VMM creates the address space and data structures required to support the new virtual machine and notifies all VxDs that a new virtual machine has been created.

Threads

Windows 95

A Windows 95 *thread* is an execution context, a register state, a stack, a priority level, and other attributes. Each non-Windows virtual machine has exactly one thread. Within the system virtual machine, there is one thread for all 16-bit Windows applications and at least one thread for each Win32 application. Win32 applications can create multiple threads if needed. The Windows kernel itself uses several threads. Although rarely necessary, a VxD can create a thread using the service VWIN32_CreateRing0Thread.

Scheduling

In Windows 3.1, the schedulable entity is a virtual machine, i.e., the scheduler allocates processor time to each VM in turn. In Windows 95, the scheduler allocates processor time among the currently active threads.

In brief, the VMM actually implements two schedulers that cooperate to determine which virtual machine should run at any given time. The two components of the scheduling system are designated as *primary* and *secondary*.

The primary scheduler uses a priority algorithm that always gives processing time to the non-suspended thread with the highest execution priority.

Priorities are adjusted up and down in response to various events that can occur. These adjustments are known as *boosts*, or *negative boosts* to decrease priority. For example, a VxD might boost a thread in response to interrupt or I/O activity.

The secondary scheduler, also known as the time-slice scheduler, periodically boosts and unboosts threads in order to balance foreground and background processing according to the user's preferences.

Because all 16-bit Windows applications run on the same thread, the Virtual Machine Manager does not preemptively schedule 16-bit Windows applications with respect to each other. However, all Win32 threads are independently scheduled along with the Windows 16-bit thread and the threads for non-Windows VMs.

If you have a good understanding of the 80x86, you might be tempted to assume that each VM runs in a different task, each with a unique task state segment (TSS). In fact, task switching using the TSS mechanisms was deemed too slow and the VMM maintains only a single TSS.

Virtual Memory

The Virtual Machine Manager provides virtual memory services for all virtual machines. Memory pages that have not been recently accessed might be swapped to disk, providing applications with an address space larger than the amount of physical memory.

Virtual Memory Mapped Devices

The Virtual Machine Manager configures the address space to restrict access to memory addresses designated as memory mapped device areas, such as the monochrome video screen at location B000h. When a program attempts to read or write such a memory address, a fault occurs. The Virtual Machine Manager dispatches the fault to the virtual device designated to handle the faulting memory.

Control Block

The Virtual Machine Manager creates a data structure called the *Control Block* for each virtual machine. In addition to accessing fields in the Control Block, VxDs can allocate space in the Control Block for private data using the **Allocate_Device_CB_Area.** Whenever a new virtual machine is created, the Virtual Machine Manager allocates a new Control Block large enough to hold the private data allocated by each VxD.

Thread Data Slots

Windows 95

VxDs that need to maintain per-thread data can allocate storage that is directly associated with VMM's thread data structures. This enables VxDs to locate data associated with a given thread without an expensive look-up operation. A thread data slot is exactly four bytes of storage at a fixed offset in each thread control structure maintained by the VMM. A VxD allocates a thread data slot with **AllocateThreadDataSlot** and frees it with **FreeThreadDataSlot**.

Virtual I/O

In addition to the ring protection levels, which control access to privileged instructions and addresses, the x86 architecture also provides for virtualization of input and output instructions.

The Virtual Machine Manager assigns privilege levels so that all access to the I/O address space, whether from V86 or protected mode, is controlled by the processor's I/O permission bit map tables. By setting bits in the I/O permission bit map, the VMM controls whether or not applications can directly access specific hardware.

VxDs call VMM services to install handlers used to virtualize I/O access to specific addresses. When an application attempts to access a device at an I/O address that is virtualized, a fault occurs and the VMM regains control. The VMM forwards the fault to the correct VxD for processing.

Virtual Machine Manager is not Reentrant

Although the Virtual Machine Manager is a multitasking operating system, most VMM services are *not* reentrant. Therefore, VxDs must serialize access to the VMM. This means that while executing a hardware interrupt handler or asynchronous timeout handler, a VxD can call only a restricted subset of VMM services. If the documentation for a service states it is "asynchronous", it is fully reentrant and can be called at any time.

VxD Overview

VxDs development requires particular care. As an adjunct to the operating system, VxDs can take over the file system and modify or disable any physical or virtual device on the computer. Test new VxDs thoroughly in as many different hardware and software configurations as possible to reduce the possibility of inadvertently destroying data. Test with the debugging kernel, available with the SDK and DDK, to take advantage of additional validation code that can be useful in identifying potential bugs.

Virtual Device Drivers are loaded as part of the Virtual Machine Manager and constitute the lowest level of the Windows operating system. Developers write VxDs to manage hardware and to extend the capabilities of Windows. For example, Virtual Devices Drivers can:

- Arbitrate contention between virtual machines seeking access to the same device
- Modify or enhance the behavior of existing hardware devices
- Create 'virtual' devices that are implemented entirely in software
- Monitor system behavior
- Provide communication between DOS virtual machines and/or Windows applications
- Implement an entirely new file system architecture
- Replace DOS TSRs, remaining resident while requiring little or no low address space
- Redirect device I/O to another device located on a network
- Provide new virtual memory services

As you can see, VxDs can be used to accomplish many different tasks. In addition, as you might expect, learning to program VxDs to accomplish such a variety of tasks is no small job. The list of services available to VxDs includes hundreds of different calls, and programming VxDs requires special care. Because VxDs operate without a safety net of protection, small slips can have far reaching consequences for the entire system. Finally, VxDs are particularly difficult to debug because they operate in an event-driven environment that makes reproducing bugs notoriously difficult.

Once past these hurdles, VxDs offer rewarding access to the "inner sanctum" of the operating system. It is here that programmers venture when the existing interfaces do not suffice, when performance is at a premium, and when it is time to push the envelope.

VxDs operate in a "habitat", or environment, that is very different from the environment to which most programmers are accustomed. Virtual devices can access and modify memory in multiple address spaces. Virtual device drivers operate interdependently, calling services exported by the Virtual Machine Manager and other VxDs. A typical VxD does not have a well defined linear flow of control. Instead, the virtual device monitors activity initiated by hardware or other software applications and responds to a variety of events using *callback* handlers that can be installed and removed dynamically.

VxD Habitat

When a VxD is running, the DS, ES, and SS segment registers are loaded with a GDT ring-0 data selector that has a base address of 0 and a limit of 4 GB. The CS register is loaded with a GDT ring-0 code selector that has the same base and limit. VxDs should never reload CS, DS, ES, or SS. FS and GS are available for use by VxDs.

Virtual devices are loaded into the same address space as the Virtual Machine Manager. This flat 32-bit address space spans the entire 4GB linear range addressable by the processor. Under Windows 3.x, this means VxDs can directly examine any memory address in any context at any time (unless that memory has been swapped to disk). Under Windows 95, memory in the private Win32 address space is only accessible within the memory context of the correct Win32 process. Determining the correct flat linear address to use in order to examine memory in a particular process address space can be tricky at times.

A VxD runs in the context of a particular virtual machine, the "current" VM. The identity of the current virtual machine might not matter when servicing hardware interrupts or examining memory. However, when a VxD needs to transfer control to routines implemented in a Windows or DOS program, it is important to be in the correct context. VxDs can schedule special events that cause handlers to be called once the desired virtual machine or thread becomes current.

Once a VxD identifies the correct linear address for a particular section of memory, accessing that memory is simple. The complications of segmented addresses are gone in a 32-bit flat address space. In C, simply cast the linear address to a pointer of the correct type, and proceed to read or write the desired data.

Ring 0 Address Space

Windows 95

VxDs can directly address the entire ring-0 address space at any time. Addresses provided by the Virtual Machine Manager or other VxDs can always be used without modification or complication.

However, when accessing memory in the private arena, keep in mind the pages mapped to this region change each time a different Win32 process becomes current. The same is true for pages in the private V86 partition and other instanced pages of the DOS arena.

Application CPU Registers

When a VxD is running, the general registers of the current VM are available in the **Client_Register_Struct ,** which is passed into most event handlers. VxDs can examine and modify the structure contents. When the virtual machine resumes execution, changes made to the **Client_Register_Struct** will be reflected in the virtual machine's registers. The **Client_Register_Struct** can also be located through the **CB_Client_Pointer** field in the VM **Control Block**.

V86 Address Space

When the processor is in V86 mode, it can address only the first megabyte (plus 64K) of the linear address space.

The first megabyte of the virtual machine address space, excluding shared DOS and BIOS pages, is instanced for each virtual machine. However, it is possible for a VxD to access memory in the first megabyte of any virtual machine, even those that are not current. Each virtual machine's 'low' memory is mapped to a 'high' linear address. The VM Control Block contains a field (**CB_High_Linear**) that specifies the linear address of the virtual machine memory. VxDs can directly access any VM's memory through its high linear address.

The handle of a virtual machine, which is passed in as an argument to most event handlers and callbacks, is a pointer to the VM Control Block. The help files include details about all of the fields in the VM Control Block.

The Virtual Machine Manager provides the **Map_Flat** and **Map_Lin_To_VM_Addr** services to translate addresses from the current virtual machine's CPU registers into linear addresses that can be used by a VxD. Addresses segment/offset forms that are not loaded into the current client register structure can be translated using real mode address arithmetic. The flat linear address can be determined by adding the **CB_High_Linear** address to the resulting 0-based byte offset.

Protected Mode Address Space

Protected mode applications (including, of course, all Windows applications) run in memory above linear address 1 MB. This memory is allocated from a common pool and appears in the flat linear address space of the VxD. However, translating an address used in protected mode is complicated by the segmentation scheme used by 16-bit protected mode applications.

Each virtual machine has a Local Descriptor Table (LDT) that defines the accessible address space for its protected mode applications. To translate a *selector:offset* address into a linear address that can be used by a VxD, add the specified offset to the base address from the descriptor as found in the correct LDT. VxDs can use the **Map_Flat** service to translate a protected mode address to a linear (ring-0) address in the current virtual machine if the selector/offset address is loaded into the current client register structure. Otherwise, VtoolsD provides the **MapPointerToFlat** function that translates an arbitrary selector, offset, and VM handle to a flat linear address.

Windows 95

Win32 applications use 32-bit flat addresses. As in the VxD environment, the base addresses of all segments whose selectors are loaded in the segment registers are zero. Under Windows 95, a pointer passed from a Win32 application to a VxD does not require any translation as long as the VxD is running in the context of that Win32 application. In order to be called in the context of a particular application, a VxD can schedule a thread event.

Virtual Memory Note

Windows 95

The Virtual Machine Manager supports virtual memory. Using secondary storage on disk, the virtual memory system creates an address space larger than the physical RAM.

Windows 95 allows VxD code and data to be paged to disk. During interrupt processing, VxDs must be careful not to access any memory that is swapped out. Use service **Page_Lock** to lock memory that is accessed at interrupt time.

If a VxD needs to access a memory location in the private arena of a Win32 application at times when the memory context of that application is not current (e.g., during hardware interrupt processing), it must create an alias for that location in the shared arena by calling **LinPageLock**, with the flags parameter set to PAGEMAPGLOBAL. **LinPageLock** returns a linear address that is globally accessible, and prevents the memory region from being swapped to secondary storage.

VxD Execution – Flow of Control

VxDs do not have a single, linear flow of control. The VxDs in a system comprise a confederation of handlers, bound together by the Virtual Machine Manager. As events occur, they are dispatched to the appropriate handler. A variety of events are triggered by hardware and software activity, including interrupts, exceptions, and faults. Other events are triggered when special conditions occur, such as time-outs, idle loops, and when synchronization objects become available.

VxDs call the Virtual Machine Manager or other VxDs to subscribe to notification of events of interest. To subscribe for notification, a VxD supplies the event dispatcher with the address of a handler to be called when the event occurs.

Windows 95 implements several domain specific interfaces at the VxD level. Examples of this architecture include VCOMM port drivers, Installable File System Drivers, and IOS drivers (VSDs, port drivers). In each case, a VxD registers a set of entry points called to perform specific services or to notify the VxD of some event.

In addition to dispatching events, the Virtual Machine Manager generates control messages before and after significant activities take place. Every VxD is required to provide a control message handler.

Loading VxDs into Memory

Loading Under Versions of Windows Prior to Windows for Workgroups 3.11

In versions of Windows prior to Windows for Workgroups 3.11, VxDs are loaded when Windows starts and are not unloaded until Windows terminates. VxDs are loaded before any applications are run and any virtual machines are initialized.

Most VxDs are loaded in response to a **DEVICE=<vxdname.VXD>** line in the [386Enh] section of **SYSTEM.INI**. It is also possible for a TSR program to force a VxD to be loaded when Windows is started.

Loading under Windows for Workgroups 3.11

Under Windows for Workgroups 3.11, VxDs can be loaded statically, as they are under earlier versions of Windows, or they can be loaded dynamically. A special VxD named VXDLDR provides services callable from applications or other VxDs that enable loading and unloading of VxDs on an “as needed” basis. This conserves system resources by requiring that a VxD be loaded only while it is in use.

The services of VXDLDR can be invoked by other VxDs, by protected mode applications, by V86 mode applications, or via Win32 API DeviceIoControl.

Loading under Windows 95 and Windows 98

Windows 95

Windows 95 supports the methods used to load VxDs under Windows 3.x, and adds a few new ones:

- VxDs named in the system registry under the key **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD** are loaded statically. Each subkey has the device name of a virtual device. For each subkey that has a string value named "StaticVxD", the data for that string value is the pathname of a static VxD to be loaded.
- The Configuration Manager, which implements the framework of the Plug and Play architecture, invokes special "device loader" VxDs that call VXDLDR to dynamically load other VxDs. The Plug and Play architecture is based on a hierarchy of protocols for detecting and enumerating devices on various system buses. Configuration information is stored in the registry and checked when the system boots up or a new device is added. See the DDK documentation for details on Plug and Play.

- Using the Win32 API call **CreateFile**, an application can force loading of a dynamically loadable VxD by specifying the file name as `"\\.\xxxx"`, where `xxxx` is the file name containing the device driver, including the file extension, which is typically .VXD.
- Dynamically loadable VxDs with the extension **.vxd** that reside in subdirectory **SYSTEM\IOSUBSYS** of the Windows tree are dynamically loaded by the I/O Supervisor at system start-up time. This mechanism is reserved for VxDs that work in conjunction with the I/O Subsystem.

Control Messages

The Virtual Machine Manager sends control messages to all VxDs as notification that various events have occurred or are about to occur. For example, control messages are sent whenever a virtual machine is created and destroyed, when Windows starts and terminates, and when protected mode applications start and end. The VxD is free to ignore any control messages that are not of interest. All control messages are fully documented in the VTOOLSD on-line help files.

Sometimes the VMM directs a control message specifically to a single VxD, rather than broadcasting it to all VxDs. The W32_DEVICEIOCONTROL control message, used to communicate with Win32 applications, is an example of such a message.

Control message handlers are the initial entry points into a VxD. All other events and callbacks of interest must be first hooked during control message handling.

The VMM broadcasts several control messages during system initialization, offering VxDs the opportunity to perform device initialization. VxDs that require per-VM data structures should allocate those structures when the **Create_VM** message is received, and can free the allocated memory when the **Destroy_VM** message is handled.

Dynamically loaded VxDs receive two messages that statically loaded VxDs do not, namely **SYS_DYNAMIC_DEVICE_INIT** (sent when the VxD is loaded) and **SYS_DYNAMIC_DEVICE_EXIT** (sent when the VxD is unloaded).

Windows 3.1

Under Windows for Workgroups 3.11, these are the only control messages that a dynamically loaded VxD receives.

Windows 95

Under Windows 95, dynamically loaded VxDs receive all broadcast control messages.

In many cases, VxDs can prevent certain activities from occurring by returning a failure code from the control message handler. For example, a VxD can prevent Windows from creating a new virtual machine by returning a failure code from the **VM_Init** message handler.

All of the control messages are enumerated in the VTOOLSD online help.

Hardware Interrupts

The Virtual Programmable Interrupt Controller Device (VPICD) dispatches hardware interrupts to registered handlers. Although it is possible for VxDs to modify the Interrupt Descriptor Table (IDT) directly, doing so is *strongly* discouraged. The services provided by VPICD should prove to be sufficient to manage hardware interrupts correctly and expediently.

VxDs can install a hardware interrupt handler by calling the **VPICD_Virtualize_IRQ** service. The name of this service is somewhat misleading, as it stems from the original conception of VxDs as a means to simulate a virtual machine environment. This service is best thought of as the means by which a VxD claims responsibility for handling a particular IRQ.

The parameter to **VPICD_Virtualize_IRQ** is a structure containing the addresses of five notification routines in the calling VxD, which VPICD calls to notify the VxD of events related to the interrupt. The most important of these events is the actual hardware interrupt event. When the specified interrupt occurs, a handler specified in the call to **VPICD_Virtualize_IRQ** is immediately called.

The VxD's operation at this point depends entirely on the nature of the virtualization being performed for the particular device. For example, a VxD can assert a virtual interrupt to a handler in the particular virtual machine that 'owns' the device and allow the V86 or protected mode program the opportunity to respond to the interrupt. This approach suffers from long interrupt latency, but might be preferable in cases where it is necessary to support an existing code base that implements ring 3 handlers.

More often, VxDs service interrupts immediately at ring 0, rather than arbitrating ownership and reflecting the interrupt to an application. For example, if a VxD is supporting a device that generates a large amount of data, the VxD might choose to buffer the data and pass it to an application at a later time. This improves performance by eliminating the time required to virtualize the interrupt, i.e. to schedule the appropriate virtual machine and dispatch to a ring 3 interrupt handler in that virtual machine.

The **VPICD_Force_Default_Behavior** service can be used to remove the virtualization of a particular IRQ.

Chapters 7 and 8 include additional information on VPICD and hardware interrupts.

Note on interrupt latency: The Virtual Machine Manager is not a real-time operating system. Interrupt latency, the time interval from the assertion of the IRQ to the invocation of the interrupt handler, is substantially shorter for a handler in a VxD than for a handler in an application or DLL. Even so, there is nothing to prevent a DOS application or another VxD from disabling interrupts for arbitrarily long intervals. Hardware that cannot tolerate occasional long latencies will not be reliable unless the software allowed to run on the system is strictly controlled. The only way to guarantee minimum response time is to poll the device status from ring 0 with interrupts disabled. This extreme measure is unacceptable in normal environments because it interferes with the operation of other devices.

Event Processing

The Virtual Machine Manager maintains an event list for each thread (for Windows 3.1, for each Virtual Machine) and one global event list. Before returning control to a virtual machine, the Virtual Machine Manager processes all events on the global event list, followed by all events for the current VM and thread. During event processing, it is safe to call VMM services, i.e., reentrancy to the VMM is not an issue.

Events can be scheduled (i.e. placed on an event list) in response to interrupts, faults, I/O operations, or programmed requests. A VxD can schedule events for itself when processing must be delayed for some reason. For example, a VxD can schedule a VM event in order to process an interrupt in the context of a specific virtual machine.

Global events can be scheduled, for example, in order to delay processing until a current interrupt handler finishes, but before control is returned to any virtual machine.

Windows 95

Priority events include a boost value added to the execution priority for the duration of the event processing.

Application time events (also known as "appy time events") are used to serialize entry to the 16-bit ring 3 Windows subsystem. VxDs that need to invoke 16-bit Windows APIs can use an appy time event to do so.

Events that have not yet been processed can be canceled.

The Virtual Machine Manager provides the following set of services to allow VxDs to create and cancel events:

```
Call_Global_Event
Call_Priority_VM_Event
Call_VM_Event
Cancel_Global_Event
Call_Restricted_Event
```

Windows 95

```
Schedule_Thread_Event
SHELL_CallAtAppyTime
Cancel_Priority_VM_Event
Cancel_VM_Event
Schedule_Global_Event
Schedule_VM_Event
```

Synchronization and Scheduling

VxDs often need to synchronize operation between multiple virtual machines in order to protect data structures from corruption. Synchronization also allows VxDs to serialize processing between multiple virtual machines.

The Virtual Machine Manager provides several synchronization services to make this possible.

Additionally, VxDs can request to be notified when certain conditions are met, such as when the system is idle or when task switching occurs.

Mutexes and the Global Critical Section

Under Windows 3.x, the Virtual Machine Manager provides a single global critical section that can be used to serialize access to sensitive data. VxDs use the critical section to prevent regions of code from being reentered. When a virtual machine claims the critical section (using the **Claim_Critical_Section** or **Begin_Critical_Section** service), the virtual machine is boosted. Scheduling is not disabled, and other virtual machines can run. However, if another virtual machine attempts to claim the critical section, the subsequent virtual machine is suspended until the owner releases the critical section using the **End_Critical_Section** service.

Windows 95

In addition to the global critical section, Windows 95 implements multiple critical sections known as "mutual exclusion" sections, called *mutexes*. A VxD calls **CreateMutex** to obtain a mutex handle and **DestroyMutex** to release it. To enter the mutex, which is analogous to claiming the critical section, a VxD calls **EnterMutex**. The analogue to **End_Critical_Section** is **LeaveMutex**. VxDs that simultaneously utilize multiple mutexes must follow a protocol to always enter them in the same order and leave them in the reverse order, so as to avoid deadlocks.

Semaphores

Semaphores provide a finer-grained level of synchronization than the critical section. When a semaphore is created, the creator specifies a *token count* that controls the number of processes that can claim the semaphore concurrently. The Virtual Machine Manager provides the **Create_Semaphore, Wait_Semaphore, Signal_Semaphore,** and **Destroy_Semaphore** services to create and use semaphores. While the critical section can be used to regulate access to DOS or BIOS, semaphores are more typically used to control access to VxD data structures, such as queues, or to implement synchronization between virtual machines.

Synchronization IDs

A limited subset of semaphore functionality is implemented by services **BlockOnId** and **SignalID**. Like semaphores, a Block ID can be used to suspend a thread until signaled by another thread. Block IDs are simpler and faster than semaphores, but do not implement a token count. Therefore, they must be used with caution to avoid race conditions. Code that blocks after initiating an action that will eventually result in signaling an ID must ensure the call to **SignalID** does not occur before the call to **BlockOnID**. Failure to do so could block the thread indefinitely.

Callbacks

Sometimes a VxD needs to be called whenever certain conditions are met. VxDs can use the following services to request callbacks when the corresponding conditions are met:

	Service	Condition
	`Call_When_Idle`	Called when all virtual machines are idle. Used for background operations.
	`Call_When_Not_Critical`	Called when a virtual device releases the critical section.
	`Call_When_Task_Switched`	Called whenever the Virtual Machine Manager carries out a task switch.
	`Call_When_VM_Ints_Enabled`	Called whenever the virtual machine enables interrupts.
	`Call_When_VM_Returns`	Called when the virtual machine executes an IRET instruction for the current interrupt. Most often used with **`Hook_V86_Int_Chain`**.
Windows 95	`Call_When_Thread_Switched`	Called whenever the Virtual Machine Manager carries out a thread switch

Interface Between VxDs and Applications

There are many ways control can pass back and forth between application software and virtual devices. In some cases, applications and VxDs establish explicit interfaces. In other cases, VxDs trap interrupts, faults, and I/O requests in order to virtualize an application's environment or to monitor its behavior.

Control Transfer into VxDs

Aside from hardware interrupt processing, control passes from applications to VxDs under the following general conditions:

- Device virtualization
- Software interrupt and fault trapping
- Explicit API services
- Hooking services of other devices

Device Virtualization

Many VxDs interact with applications through software emulation of a physical device, or *virtualization*. From the perspective of an application, the device in question can be accessed by reading or writing memory, by reading or writing I/O addresses (ports), or by responding to interrupts. In all these cases, a VxD can use VMM services to intervene between the application and the physical device.

The virtual device intercepts the I/O instructions, such as IN and OUTSB, by installing a handler for a specific port address with the **Install_IO_Handler** service. Trapping can be disabled system wide or for specific VMs using the **Disable_Global_Trapping** and **Disable_Local_Trapping** services.

VxDs can use the **VPICD_Set_Int_Request** service to simulate a hardware interrupt in a virtual machine.

In some cases, the VxD will act as a 'buffer' between an application and a device actually installed in the computer. In this case, the VxD will receive hardware interrupts directly, and can control the hardware as needed. Data can be buffered if necessary and forwarded to the application using simulated interrupt and input/output services.

For more information on handling and virtualizing hardware interrupts, refer to the help files for VPICD services.

VxDs can virtualize Direct Memory Access (DMA) channels by using services exported by the VDMAD device. Use the **VDMAD_Virtualize_Channel** service to register a callback that will be called whenever the virtual state of the DMA channel is changed as a result of I/O carried out in a virtual machine.

Software Interrupt and Fault Trapping

Virtual devices can install handlers called whenever specified software interrupts are executed in a virtual machine. For example, a VxD could intercept calls to DOS by installing an INT 21h in V86 mode. Similarly, a VxD can intercept faults that occur in applications.

VxDs can hook interrupts and faults in order to monitor the behavior of applications. In this case, the VxD would forward the interrupt to the original handler. In other cases, VxDs will override the default behavior of software interrupts in much the same way a real mode TSR might.

VxDs can hook software interrupts that occur in V86 mode by using the **Hook_V86_Int_Chain** service. Hardware interrupts simulated into the virtual machine are also hooked using this service.

Software interrupts occurring in protected mode can be hooked by creating a protected mode callback (**Allocate_PM_Call_Back** service) and installing the callback into the protected mode interrupt vector using the **Set_PM_Int_Vector** service.

In some cases, a VxD will use the **Install_V86_Break_Point** service in order to trap all execution at a particular address. This technique might be used, for example, to intercept not only INT 21h calls, but also calls to DOS made with a PUSHF/ CALL sequence to the address of the INT 21h handler. Because breakpoints consume resources and make it difficult to chain to the previous handler, a better alternative is to use **Allocate_V86_Call_Back** and **Set_V86_Int_Vector**.

The **Hook_VMM_Fault** service is used to catch interrupts and faults that occur in ring 0. Faults that occur in V86 or protected mode are hooked using the **Hook_PM_Fault Hook_V86_Fault** services.

Other related services that can be used include:

```
Get_Fault_Hook_Addrs

Get_NMI_Handler_Addr

Hook_Invalid_Page_Fault

Hook_NMI_Event

Set_NMI_Handler_Addr

Unhook_Invalid_Page_Fault
```

Explicit API Services

In many cases, virtual devices will offer explicit entry points that can be called by other programs. A VxD can create interfaces called by other VxDs, by V86 (DOS) programs, or by protected-mode programs such as Windows and DOS-extended applications.

Services for V86 and PM applications

The Virtual Machine Manager automatically creates linkage for applications to call special VxD handler routines identified as protected-mode or V86 API handlers in the VxD Device Descriptor Block. These interfaces are described in detail in the C Framework and Class Library chapters.

The Virtual Machine Manager also provides services that allow virtual devices to create additional handlers using callbacks, such as **Allocate_PM_Call_Back** and **Allocate_V86_Call_Back.** These services return an address that can be called by protected-mode or V86 programs in order to transfer control to the VxD handler.

The protection mechanisms of the x86 architecture prevent direct calls between software running at different privilege levels. Although call gates could be used to effect a transfer, the Virtual Machine Manager places instructions into the V86 or PM address spaces that cause faults when executed. The Virtual Machine Manager recognizes faults at these addresses as interface calls, and dispatches to the appropriate VxD.

Services for Win32 Applications

Windows 95

To provide services for Win32 applications, VxDs handle control message W32_DEVICEIOCONTROL. A Win32 application obtains a handle to a VxD using API **CreateFile** and then calls it with **DeviceIoControl**. This causes a control message to be sent to the VxD in question. See Chapter 9 for details on how to code this.

Services for Other VxDs

Services to other VxDs are made available through the virtual device's **VXD_Service_Table,** which is created when the VxD is loaded. When another VxD needs to call such a service, it issues an INT 20h (in ring 0, of course).

The 32-bit value immediately following the INT 20h instruction identifies the service being called. The first 16 bits form the device ID. Each VxD that provides services to other VxD must have a unique device ID (the ID of the Virtual Machine Manager is 0001). The next 16 bits form the service number, which is a zero based index into the called VxDs service table. The most significant bit of the service number is set to one if the transfer is a jump, and to zero if the transfer is a call.

The VMM replaces the INT 20h/dword with a call to the target service, thereby "snapping" the dynamic link. This link snapping improves performance on future calls. Calls to VxD services are always made indirectly through the service table of the called VxD.

Dynamic VxDs that provide services to other VxDs cannot be unloaded once a call to the target VxD has been "snapped," since the reference would become invalid if the target driver was removed from memory.

Hooking Services Provided by Other VxDs

VxDs can also intercept calls to the Virtual Machine Manager or other VxDs. This allows a VxD to modify the behavior of an already loaded device, either in whole or in part. The VMM provides three services that allow one virtual device to replace the services of another:

```
Hook_Device_Service

Hook_Device_PM_API

Hook_Device_V86_API
```

In addition, the VTOOLSD wrapper library provides **Hook_Device_Service_Ex**, which, like **Hook_Device_Service**, hooks a service of the VMM or other VxD. The handler for the extended service returns a Boolean indicating whether or not the original service address should be called or not. This allows a VxD to monitor calls to a service and to conveniently invoke the original service. See **Call_Previous_Hook_Proc** also.

Calling Application Level Code from VxDs

Sometimes a VxD needs to call application code directly. Again, the protection mechanisms of the x86 make the process indirect.

To call entry points in Win32 applications or DLLs, a VxD uses **Asynchronous Procedure Calls**, discussed in Chapter 9. To call application code other than Win32 applications and DLLs, the mechanism for transferring control into a ring 3 program is the set of nested execution services provided by the Virtual Machine Manager.

In order for a VxD to transfer control to a ring 3 address (V86 or 16-bit protected mode), the desired routine must be in the currently active virtual machine. If necessary, the VxD should schedule an event in the desired VM. This installs a callback that will be invoked when the specified event conditions are met.

Once the virtual device is running in the context of the desired virtual machine, the VxD can invoke the nested execution services. These services are used to modify the stack of the application and to 'call' the application by transferring control to a specified address. When the application 'returns' (usually via IRET or RET instructions), control is transferred back to the VxD. The nested execution services are *not* available while processing a hardware interrupt.

The client's registers must be saved and restored so that execution can continue correctly after the nested execution completes.

The VMM provides services used to save and restore the client registers (**Save_Client_State** and **Restore_Client_State**). The on-line help for **Begin_Nest_Exec** contains an example demonstrating how to use a nested execution block to call a DOS service.

Other calls that are used to establish and execute nested execution blocks include:

```
Begin_Nest_V86_Exec

End_Nest_Exec

Exec_Int

Resume_Exec_State

Set_PM_Exec_Mode

Set_V86_Exec_Mode
```

The VMM also provides a set of related services that can be used to create and modify the execution context, including:

```
Build_Int-Stack_Frame

Simulate_Far_Call

Simulate_Far_Jmp

Simulate_Far_Ret
```

```
Simulate_Far_Ret_N

Simulate_Int_N

Simulate_Iret

Simulate_Pop

Simulate_Push
```

Restrictions on Calling 16-bit Ring 3 Code from VxDs

Because VxDs are event driven, their execution is asynchronous with execution of application level code, including Windows applications. This is important to consider when making nested execution calls, because Windows APIs are not reentrant. The only exceptions to this rule are **PostMessage** and **PostAppMessage**, which are guaranteed to be reentrant.

Using nested execution services, it is safe to invoke an entry point in a custom 16-bit DLL as long as that code is fully reentrant and does not call Windows APIs (other than the above mentioned).

Synchronization restrictions do not apply when the VxD is executing in the context of an Application Time Event handler. “Appy time” events are discussed in Chapter 9.

Posting Messages to Windows Applications from VxDs

Windows 95

Windows 3.1

Under Windows 95/98, a VxD can call service **SHELL_PostMessage** to post a message to the window of a 16-bit or 32-bit Windows application. This eliminates the need to use nested execution services for this common operation. The application or DLL must have previously communicated the window handle (HWND) to the VxD.

SHELL_PostMessage is not available under earlier versions of Windows.

Additional coordination is required to enable a VxD to invoke **PostMessage** (or **PostAppMessage**). The VxD must locate the address of the **PostMessage** entry point in the system VM. The simplest technique requires either a ‘helper’ Windows application that passes the address of the **PostMessage** entry point to the VxD during initialization, or one that can be signaled to perform the **PostMessage** on behalf of the VxD.

Any attempt to perform nested execution into 32-bit Windows code will fail. Nested execution into 32-bit code of virtual machines other than the system virtual machine is allowed.

VxD Structure

Windows 95

The VxD loader in the Virtual Machine Manager expects a specific file format. VxDs under Windows 95 are generally named with an extension of **.VXD**. Under Windows 3.x, VxDs were by convention named with the extension **.386.** VxDs are stored in a file format known as LE.'

Every VxD has a Device Descriptor Block (DDB). The DDB contains enough information for the Virtual Machine Manager to determine the initialization order of the device, to find the control message handler, and to establish linkages to applications and other VxDs that might need to call services provided by the virtual device.

The LE' file format controls how code and data blocks will be loaded into ring-0 memory.

Device Descriptor Block

The Device Descriptor Block is identified in the file as the first (and typically only) exported entry point. The DDB fields are:

Field Name	Description
`Next`	Used by the VMM to link all of the DDBs in the system.
`SDK_Version`	Identifies what version of the DDK was used to build the VxD.
`Req_Device_Number`	Device ID used by other VxDs, V86, and PM applications to find the VxD and to call VxD services. If no services are exported or all services are exported using Vendor strings or other means, then the Device ID should be set to UNDEFINED_DEVICE_ID.
`Dev_Major_Version`	Major version number.
`Dev_Minor_Version`	Minor version number.
`Flags`	Various flags.
`Name`	Device name (must be blank-filled to 8 characters).
`Init_Order`	Initialization order. The order should be expressed in terms of the initialization order of other devices. If the device does not depend on having other devices loaded first nor need to be loaded prior to other specific devices, then UNDEFINED_INIT_ORDER should be specified.
`Control_Proc`	Offset (linear address) of control procedure. This procedure is called whenever a control message is available for the device.
`V86_API_Proc`	Offset (linear address) of API procedure to be called when to implement V86 services.
`PM_API_Proc`	Offset (linear address) of API procedure to be called to implement Protected Mode services.

V86_API_CSIP	Address (CS:IP) stored by VMM which is called by V86 programs. These calls are reflected to the V86_API_Proc (see above).
PM_API_CSIP	Address (CS:IP) stored by VMM which is called by Protected Mode programs. These calls are reflected to the V86_API_Proc (see above).
Reference_Data	Reference data from real mode. (Holds DRP address for IOS drivers.)
VxD_Service_Table_Ptr	Pointer to VxD Service Table (if present). This table identifies services that can be called from other VxDs.
VxD_Service_Table_Size	Number of services in the VxD Service Table (if any).

Segmentation

On disk in the LE file format, a VxD is partitioned into multiple memory objects, each with its own attributes (e.g. readable, writable, executable, discardable, etc.). When a VxD is loaded, either statically or dynamically, each of its memory objects is examined and classified. The VMM and VXDLDR recognize a fixed set of attribute combinations, each of which maps to one of the segment types shown in the following list:

Locked Code Locked Data	Remain resident in memory
Init Code Init Data Init Message	Discarded after initialization
Pageable Code Pageable Data	Can be swapped to disk
Static Code Static Data	Segments of dynamic VxD that remain resident when VxD is unloaded
Locked Message Pageable Message	Contain messages
Debug-Only Code Debug-Only Data	Loaded only if debugger is present

Windows 95

If a VxD contains a memory object whose attributes the VMM or VXDLDR does not recognize, the VxD is not loaded. Note that a driver designed for Windows 95 cannot be loaded on a Windows 3.1 system if it contains any segments other than LOCKED and INIT.

Real Mode Initialization

There is one additional segment defined for VxDs. The REAL_INIT segment is used for code and data that are required during real mode initialization, called in real mode before the Virtual Machine Manager begins running. The contents of this segment are not available after the Virtual Machine Manager starts.

Summary of Services Available to VxDs

There are nearly 1,000 documented services available for VxDs. The Virtual Machine Manager implements a dynamic link mechanism that enables a VxD to bind to any of these services at run time. Services are identified by a 16-bit Device ID and a 16-bit Service Number within each device. Linkage to services of the VMM is made using the same mechanism.

Most VxD-level services are defined to use register-based calling conventions, although many of the new services for Windows 95 use C-style stack-based arguments. The VTOOLSD libraries provide translation "wraps" that allow any service to be called directly using standard 'C' syntax and calling conventions. For services that accept and return arguments in registers, the wrapper moves arguments from the stack to registers on input and from registers to memory on output. For C-callable services, the wrappers provide dynamic linkage and strong type checking.

The following table lists the general categories of services provided by the Virtual Machine Manager and standard VxDs. Complete documentation is provided in the on-line help files.

The Virtual Machine Manager provides the following categories of services:

Memory Management	I/O Trapping
Interrupts	Nested Execution
Breakpoints	Scheduling
Events	Timers
Faults	Protected-Mode Execution
Information	Initialization
Linked Lists	Error Conditions
Debugging	Other Miscellaneous

In addition, there are a number of standard VxDs that provide additional services. Complete documentation is available in the help files.

The standard Windows installation includes the following virtual devices that provide services to other VxDs:

Block Device	Cache Device
Communications Device	Configuration Manager
Display Device	DOS Manager
DMA Device	EBIOS Device
Fast Disk Device	Floppy Disk Device
Installable File System Manager	Interrupt 13h Device

I/O Supervisor	Keyboard Device
Math Coprocessor Device	Mouse Device
MS-DOS Manager Device	MS-DOS Network Device
Multimedia Device Loader	Network Device (NDIS)
Page Swapping Device	PCMCIA Device
Performance Monitor	Power Management Device
Programmable Interrupt Controller (PIC) Device	Shell Device
Sound Device	Timer Device
V86-Mode Memory Manager	Virtual Machine Polling Device
VWIN32	VXDLDR

Where to Get More Information

There are few good sources of information about virtual device drivers. The Windows Device Driver Kit, available from Microsoft as part of the Microsoft Developer's Network Level II program, includes a large number of assembly-language examples along with documentation and header files designed for the assembly language programmer. The MSDN Level II program also includes access to Microsoft technical support for up to 10 "incidents".

Two books that you might find valuable are *Writing Windows VxDs and Device Drivers* by Karen Hazzah (R&D Technical Books) and *Systems Programming for Windows 95* by Walter Oney (Microsoft Press).

There are several on-line forums dedicated to discussion of Windows device drivers. On the Internet, you can point your news reader to the USENET group **comp.os.ms-windows.programmer.vxd.**

Compuware maintains a World Wide Web site for DriverStudio tools at http://www.compuware.com/products/driverstudio/ds/.

3 QuickVxD

Starting a New Project with QuickVxD

QuickVxD is a form-based utility that works with the VTOOLSD libraries to automatically generate skeletons of VxDs. The goal of QuickVxD is to give you a fast start on a new VxD project without being distracted by the low level details of VxD architecture. While it is *not* a comprehensive visual programming environment, it is a useful tool for creating the basic structure of a VxD.

QuickVxD launches a VxD project by creating three files that form a foundation upon which you build a VxD. These files are:

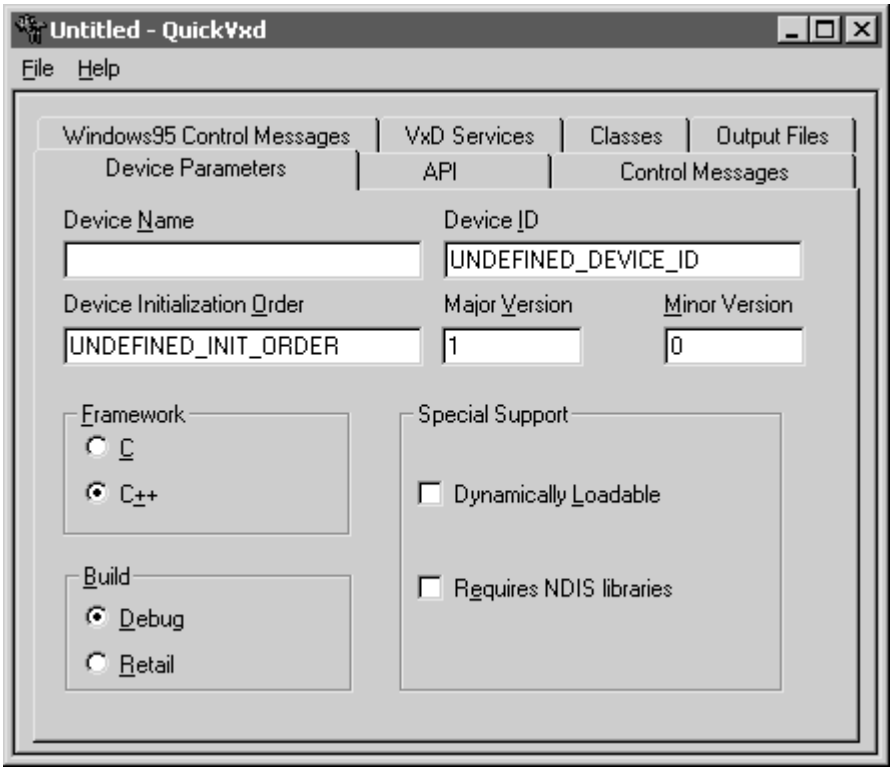
- An include file
- A code module
- A make file

The source files include the basic components you need to start a VxD, including skeletons for control message handlers, API entry points, and VxD services. In addition to function skeletons, QuickVxD generates code that defines symbols, sets up parameters for the make system, and declares classes. After QuickVxD generates the source files, you add code in the function bodies of the various handlers that have been generated and then invoke a make utility to build the VxD.

This chapter describes the version of QuickVxD distributed with VTOOLSD for Windows 95. QuickVxD for Windows 3.1 has slightly different screens, reflecting an earlier interface and absence of options introduced with Windows 95.

Specifying VxD Parameters

QuickVxD is implemented using a set of tabbed dialog boxes. Each dialog contains a set of controls that enable you to specify the characteristics of a VxD. This section explains the purpose of each control and how it affects the output file.



Device Parameters

The tab labeled **Device Parameters** selects a dialog containing controls that allow you to specify the basic parameters of the VxD.

Device Name

Use the edit control labeled **Device Name** to enter the name of the VxD you are building. The name can be up to eight characters; can contain only letters, numbers, and underscores; and must not begin with a number. The name should be unique to your device, and should generally *not* use the Microsoft convention of starting with a 'V' and ending with a 'D'. Instead, use a brand name or some other unique identifier to distinguish your device.

The Device Descriptor Block,declared in the output code module, contains a copy of the device name. In addition, the device name serves as the default name for the output files.

Framework: C or C++

The radio buttons in the group box labeled **Framework** select one of the two VxD framework styles provided by VTOOLSD.

The C Framework provides structured processing of control messages, support for V86 and protected mode entry points, as well as a complete set of wrapper functions for calling VMM services. See Chapter 6 for more information.

The C++ Framework is based on a set of classes that abstract the VxD environment. You derive classes from the base classes **VDevice**, **VVirtualMachine,** and **VThread** to provide basic control message processing and programmed entry points. Other classes in the class library access a broad variety of the VMM/VxD environment.

If you specify a device name on the QuickVxD form when you click on the C++ radio button, QuickVxD automatically inserts names for your device class and virtual machine class. You are free to modify the default names if you choose. QuickVxD automatically generates the class definitions in the output include file.

For more information about the C++ Framework, refer to the chapter on the VTOOLSD class library.

Device ID

You use the edit control labeled **Device ID** to input the number or symbol that identifies the device. A device ID is *required* if your VxD supplies services to other VxDs. Port drivers and layered block device drivers do not require a device ID.

The ID can be entered symbolically (e.g. VDD_DEVICE_ID), but must resolve to a 16-bit value when compiled. The string entered must be a valid C constant expression.

Arbitrary values should not be used because possible conflicts with existing VxDs. See Appendix D for information on how to contact Microsoft to obtain a unique ID value.

For test purposes only, you can use the string **NUMEGA_TEST_ID**.

The device ID is stored in the Device Descriptor Block and in the VxD file header.

Device Initialization Order

Use the edit control labeled **Device Init Order** to input the string that specifies the point during Windows initialization at which your VxD must be initialized. If your VxD has no special initialization dependencies, use the default value, UNDEFINED_INIT_ORDER. Otherwise, the value should be expressed relative to a known initialization order constant. Initialization order constants, found in **VMM.H**, are of the form xxxx_INIT_ORDER, where xxxx stands for the device name.

For example, if you require that the virtual display device be initialized prior to the VxD you are building, set your init order string to VDD_INIT_ORDER + 1.

The device initialization order is included in the Device Descriptor Block

Version

The edit controls labeled **Major Version** and **Minor Version** allow you to specify the version of your device. By default, they are 1 and 0 respectively.

Only enter numeric values from 0 to 255 in these controls.

The version numbers are included in the Device Descriptor Block

Debug and Retail

VTOOLSD provides two versions of its object libraries, Debug and Retail. The debug libraries, whose names end with the letter 'd', contain code that emits messages when questionable conditions arise.

The radio buttons labeled **Debug** and **Retail** select which version of the libraries are used to build the VxD. If you choose Debug, QuickVxD emits a definition to the make file that selects the debug libraries and defines symbol "DEBUG" for all compilations.

Dynamically Loadable

Windows 95 supports dynamically loadable VxDs. Check the box labeled **Dynamically Loadable** to cause QuickVxD to emit a symbol definition to the make file that results in building a dynamically loadable VxD.

NDIS Support

Check the box labeled **Requires NDIS Libraries** if the VxD you are building uses functions from the NDIS (Network Device Interface Specification) wrapper. Checking the box causes QuickVxD to emit definitions to the make file that instruct the linker to link in the appropriate NDIS library.

Control Messages

There are two tabs that select dialogs containing check boxes for the control messages the system sends to VxDs: **Control Messages** and **Windows 95 Control Messages**.

For the C Framework, selecting a message results in generation of code in the function **ControlDispatcher** and a skeleton for the function that handles the message.

For the C++ Framework, selecting a message results in generation of a declaration and a skeleton for a member function. Some messages are passed to the device object and others to the appropriate virtual machine or thread object. Dispatching to the message handlers is done transparently by the internals of the VTOOLSD class library.

Application Program Interface

V86 Mode Entry Point and Protect Mode Entry Point

Windows provides a standard mechanism for applications to call VxDs. A VxD can export handlers as the V86 Mode Entry Point or Protect Mode Entry Point to provide services to applications. The argument to these handlers is a pointer to a structure containing the application's register state. The handler can perform any operation and can modify the calling application's register state.

Check the **V86 Mode Entry Point** check box if your VxD will provide this service to V86 mode applications. Check the **Protect Mode Entry Point** check box if your VxD will provide this service to protected mode applications.

For the C Framework, checking the box causes a function skeleton to be generated for the entry point. The function name is **V86_Api_Handler** for V86 mode and **PM_Api_Handler** for protected mode.

For the C++ Framework, the entry points appear as virtual member functions **V86_API_Entry** and **PM_API_Entry** in the device class.

Chapter 6 explains the method that application code uses to obtain an address that can be called to invoke the API entry point.

Vendor Specific Entry Points

Prior to Windows 95, a VxD was required to have a unique device ID in order for its Protected Mode or V86 Mode API to be accessible. Windows 95 allows applications to locate a VxD's API entry points using only the VxD name.

For Windows 3.x, VTOOLSD provided an alternative method to enable applications to call VxDs without requiring a device ID. This method takes advantage of a convention defined in the DOS Protected Mode Interface (DPMI) specification. This method is still supported, but it is no longer recommended. For VxDs that operate under Windows 95, it is preferable to use the standard mechanism above described.

An application can query for the presence of a particular service provider by loading DS:(E)SI with a pointer to a unique string, then issuing INT 2Fh/AX=168Ah. If that service provider (i.e., your VxD) is present, it responds by clearing the AL register and returning a call back address in ES:(E)DI. Applications can call the address to invoke an entry point handler in the VxD.

QuickVxD and the VTOOLSD libraries supply all the linkage. Just check the appropriate box and supply a unique string that applications can use to locate your VxD. After output is generated, you can add code to the entry point handler function(s) in the code module.

Vendor Specific Entry Points can be called by V86 and protected mode DOS and Windows applications. However, Win32s applications cannot call an entry point obtained via INT 2Fh/AX=168Ah, because the system virtual machine runs a 16-bit DPMI client.

If you check either of the **Vendor Specific Entry Point** check boxes, QuickVxD requires you provide the unique string to identify your VxD before generating output.

If the C Framework is selected, checking **Vendor Specific Entry Point** causes the following output:

- In the include file, a **#define** for the string constant.
- In the code module, storage for an instance of the string constant.
- QuickVxD generates code in **OnDeviceInit** to initialize a V86 entry point, and in **OnBeginPMApp** to initialize a protected mode entry point.
- In the code module, skeletons for the entry point handlers.

If the C++ Framework is selected, then checking **Vendor Specific Entry Point** causes the following output:

- In the include file, a **#define** for the string constant.
- In the include file, declaration of classes derived from class **VV86DPMIEntry** and/or **VPMDPMIEntry**.
- In **OnDeviceInit** !WIN40! (or **OnSysDynamicDeviceInit** for dynamically loaded devices), invocation of constructors for entry point classes.
- In the code module, constructors for entry point classes and skeletons for the entry point handlers.

One important note about protected mode entry points: Remember that protected mode applications include DOS extended applications that can run in DOS boxes as well as Windows applications. If you want your protected mode entry point to be accessible only from Windows applications, you must edit the code module that QuickVxD produces. Move the initialization of the protected mode entry point from **OnBeginPMApp** to **OnDeviceInit**. Conversely, if you want DOS extended apps to be able to call your protected mode API, but you want to exclude Windows apps, add a call to **Test_Sys_VM_Handle** to **OnBeginPMApp** and do not initialize the protected mode entry point if it returns TRUE.

VxD Services

Select the dialog tab labeled **VxD Services** to enter C prototypes for service functions your VxD provides to other VxDs. Enter strings on the edit line and press the **Add** button.

To remove previously entered service prototypes, highlight the service line and press the **Remove** button.

For the C Framework, QuickVxD generates the code to export the function and generates a function skeleton.

For the C++ Framework, QuickVxD generates the code to export the function, and makes it a static member of the device class.

The first service provided by your VxD should be a **Get_Version** function; QuickVxD generates this automatically. This allows other devices to determine whether your VxD is loaded. The Virtual Machine Manager returns an error to the caller if service 0 is called for a device not loaded. VxDs that call services provided in other VxDs that cannot be loaded correctly should always call service 0 before calling any other service.

Classes

Device Class

The edit control labeled **Device Class** is used to input the name of the class that serves to abstract the virtual device you are building. QuickVxD enables the control only when you select the C++ Framework.

The member functions of the device class include handlers for a subset of the control message handlers, as well as the Protected Mode and V86 Mode API entry points for the VxD and VxD services.

If you have already entered a device name when you press the C++ radio button and you have not at that point entered a device class, QuickVxD inserts a default name for the device class based on the device name. You are free to modify the suggested name if you desire.

For more information about the device class, refer to the chapter on the VTOOLSD class library.

Virtual Machine Class

Use the edit control labeled **Virtual Machine Class** to specify the name of the class that abstracts virtual machines for your VxD. The role of the Virtual Machine class in the C++ Framework is to provide control message handling and allow you to implement behavioral differences between different kinds of VMs. See the VTOOLSD class library documentation for more details.

Thread Class

Use the edit control labeled **Thread Class** to specify the name of the class that abstracts threads for your VxD. The roles of the Thread class in the C++ Framework are to provide control message handling and to allow you to implement behavioral differences between different kinds of threads. See the VTOOLSD class library documentation for more details.

Saving the VxD Specification to a File

QuickVxD allows you to save the specification of a VxD in a disk file. The default file extension is .QVX.

To save a VxD specification, choose **Save** or **Save As** on the **File** menu.

You can load a previously saved VxD specification by choosing **Open** on the **File** menu.

If you exit QuickVxD, load a .QVX file, or select **New** on the **File** menu, QuickVxD will prompt you to save the current specification if it has been modified.

Generating the VxD Skeleton

To generate the output files, select the tab labeled **Output Files**. When you press the button labeled **Generate Now**, QuickVxD first verifies the device parameters you have selected. QuickVxD enforces the following rules:

- The device name must be valid.
- The device ID must contain no invalid characters.
- If either Vendor API box is checked, the Vendor ID string must not be empty.
- If there are VxD services, then the device ID must be defined.
- If Dynamically Loadable is selected, QuickVxD checks if the control messages SYS_DYNAMIC_DEVICE_INIT and SYS_DYNAMIC_DEVICE_EXIT are selected.
- In addition, QuickVxD will issue a warning if no control messages are selected.

- QuickVxD generates three files:
  - ◊ **An include file**. This contains definitions of symbolic constants that VTOOLSD uses to construct the VxD. In addition, it includes definitions for exported VxD services. Include files for VxDs that use the C++ Framework contain definitions for various classes.
  - ◊ **A code module**. This contains skeletons of functions that handle control messages, provide VxD services, and provide application entry points. Code modules for VxDs that use the C Framework contain the dispatcher for control messages.
  - ◊ **A make file**. This defines parameters that instruct VTOOLSD how to compile and link your VxD. It references make files in the INCLUDE subdirectory of your VTOOLSD directory.

Before generating the files, QuickVxD displays the file names in a dialog, and allows you to change them. If you have saved the .QVX file for the current VxD specification, the directory in which you saved it is the default directory for the output files. If the output files already exist, QuickVxD will prompt you before overwriting them. The files must not be write-protected or in use by another program.

Next Steps

Once you have successfully generated the output files for a VxD, you can build it using NMAKE (with MSVC) or MAKE (with Borland C). The next chapter explains how to do this. After confirming that your new VxD builds correctly, you can use any standard text editor to add code to the skeleton sources generated by QuickVxD.

QuickVxD cannot currently update source files once they have been generated. If you need to add code to an existing driver, you might find it convenient to run QuickVxD to generate a new VxD source module and extract pieces from the generated source code to insert into your existing driver.

4 Building and Debugging VxDs

This chapter discusses the tools and procedures that are available to compile, link, load, and debug VxDs. Chapters 6, 7, and 8 describe the code you write to make your VxD do something useful.

To get the most from this chapter, we recommend you first look at one of the example VxDs provided with the product or generate your own simple driver with QuickVxD. By doing so, you will familiarize yourself with the component files needed to build VxDs.

The issues discussed in this chapter are common to both C and C++ development.

Building a VxD

VxD development in C or C++ requires using a set of tools from several vendors that were not originally designed to work together to produce VxDs. VTOOLSD knits all of these tools together as seamlessly as possible using a set of *make files*. If you are not familiar with using utilities such as **nmake** to build programs, do not worry. In many cases, you will not need to make significant changes to the make files constructed by QuickVxD. Most choices are controlled by simple option definitions.

Make Utilities

If you are using a Microsoft compiler, the make utility you will use to build VxDs is called **nmake**. The corresponding utility for the Borland compiler is **make**. Each is found in the BIN subdirectory of the respective compiler installation. For ease of use, we recommend you put this subdirectory on your path.

If you are using the Microsoft compiler, run VCVARS32.BAT from the BIN subdirectory of the compiler installation before running **nmake**.

For information on building VxDs from the Microsoft Developer Studio environment, see the online help under **Technical Notes**.

The VTOOLSD Automated Make System

If you have used QuickVxD to construct a new VxD project, or you are connected to one of the example directories, building a VxD is simple. The following line is all that is required:

Microsoft: nmake /f <*makefile name*>

Borland: make /f <*makefile name*>

The make file name is generally either the default name MAKEFILE (in which case the /f switch is not needed) or the device name with **.MAK** as the extension, e.g. **TEST.MAK**. The Borland tool does not require you to specify the .MAK extension.

If no errors occur, the VxD will be compiled and linked with the VTOOLSD libraries to generate an output file (e.g. **TEST.VXD**) ready for loading. The following conditions must be met:

- The make utility, **nmake** or **make**, must be on your path or fully specified.
- The environment variable VTOOLSD must point to the directory where VTOOLSD is installed.
- **USER.MAK** must specify the correct paths for the 32-bit compiler and linker installed on your system.

Older versions of Microsoft's **nmake** have a bug that results in a message that the "command line is too long." If you see this message when attempting to build a VxD, you will need to use a more recent version of NMAKE.

Contents of the Make File

The make file is designed to be very simple. The basic idea is you simply set the values of a few predefined symbols, then include two special make files provided by VTOOLSD. The make utility does the rest of work.

Here is an example make file. This file (**SIMPLE.MAK)** can be found in the VTOOLSD **EXAMPLES\C\SIMPLE** directory. **SIMPLE.MAK** was generated by QuickVxD. First, a quick look at the entire file:

```
# SIMPLE.MAK - makefile for VxD SIMPLE

DEVICENAME = SIMPLE

FRAMEWORK = C

DEBUG = 1

OBJECTS = SIMPLE.OBJ

TARGET = WIN95
```

```
!include $(VTOOLSD)\include\vtoolsd.mak

!include $(VTOOLSD)\include\vxdtarg.mak

SIMPLE.OBJ:SIMPLE.C SIMPLE.H
```

Now, let us examine **SIMPLE.MAK** line by line.

The first line is a comment line - the '#' character introduces comments in the make file:

```
# SIMPLE.MAK - makefile for VxD SIMPLE
```

DEVICENAME is used to specify the name of the device:

```
DEVICENAME = SIMPLE
```

FRAMEWORK selects whether the C Framework (**C**) or C++ Framework (**CPP**) should be used:

```
FRAMEWORK = C
```

DEBUG specifies this build should include debugging information:

```
DEBUG = 1
```

OBJECTS is a list of the object files which must be linked together to build the VxD. The SIMPLE VxD requires only one object file:

```
OBJECTS = SIMPLE.OBJ
```

TARGET specifies the oldest operating system on which the VxD is to run:

```
TARGET = WIN95
```

In this case, the VxD can run on Windows 95 or Windows 98. Other valid specifiers are WIN31, WFW311, and WIN98. The effect of this setting is to screen out prototypes of system services not available on older operating systems.

If you are using the Borland compiler, please see the following instructions on specifying object file names.

The make file loads **VTOOLSD.MAK** to locate the tools needed to build the VxD and to define the compiler switches and rules used to compile the source files. In turn, **VTOOLSD.MAK** loads **USER.MAK** to determine file paths specific to your system:

```
!include $(VTOOLSD)\include\vtoolsd.mak
```

VXDTARG.MAK Defines the rules for creating the VxD from the object files:

```
!include $(VTOOLSD)\include\vxdtarg.mak
```

The make file ends with a dependency list that enables **nmake** to keep the VxD properly up to date when source files are changed:

```
SIMPLE.OBJ:SIMPLE.C SIMPLE.H
```

Summary of MAKEFILE Options

The following table summarizes the symbols recognized by the VTOOLSD make system. These options are described in further detail later in this chapter.

Symbol	Meaning	Default	Choices
ASMFLAGS	Define arguments to the assembler that will replace the standard arguments provided by the VtoolsD make system		
BROWSE	If set to 1, create a .BSC browse database (MS compilers only)	0	0, 1
CFLAGS	Flags for compiler command line (use caution in overriding defaults!)	(see .mak file)	
COPTFLAGS	Select C/C++ optimizations. This can be particularly useful to disable optimizations during DEBUG builds		
DEBUG	Selects Debug or Retail build	0 (retail)	0, 1
DEVICENAME	Names VxD (up to eight characters)	none	DEVNAME
DEVICETYPE	Select VXD or PORTDRIVER build	VXD	VXD, PORTDRIVER
DYNAMIC	Selects build of dynamically loadable VxD	0 (static)	0, 1
FRAMEWORK	Selects language based framework	none	C, CPP
NDIS	Selects linking with NDIS wrapper library	0	0, 1
NOFILTER	Suppress filtering of extraneous link and editbin messages	none	
OBJECTS	List of object file names	none	
OBJPATH	Path used to store object files	. (current directory)	
SOURCEPATH	Path used to locate source files	. (current directory)	
TARGET	Selects libraries for Windows 3.1 or Windows for Workgroups 3.11	WIN95	WIN31, WFW311, WIN95, WIN98
USER_COMMENT	Comment for DEF file	VxD devname (VTOOLSD)	
USER_LIB	List of extra libraries to search during link	none	
XFLAGS	Extra flags for C/C++ compilation (appended to CFLAGS)	none	
XAFLAGS	Extra flags for ASM compilation (appended to ASMFLAGS)	none	
XLFLAGS	Extra flags for link step	none	

In addition to the above list, there are several more options available by including file INCLUDE\VDBGREL.MAK. See examples\cpp\FANCYMAK for an example of how to use these advanced features.

Detailed Description of MAKEFILE Options

The VTOOLSD make system is designed to hide the complexity of the build details for most projects, while at the same time providing fine tuned control over most options.

ASMFLAGS

ASMFLAGS is used to define arguments to the assembler. The arguments defined by the ASMFLAGS option will replace the standard arguments provided by the VtoolsD make system. Also see **XAFLAGS**.

BROWSE

BROWSE is used to select whether or not a browse database (.BSC file) is built.

BROWSE = 0	Do not build browse database (default).
BROWSE=1	Build the browse database.

CFLAGS

CFLAGS is used to over-ride the command line options passed to the compiler. Please refer to the appropriate **.MAK** file to review the default options used by the VTOOLSD build system before changing this value. For example:

```
CFLAGS = -RT- -x- -c -Oi -Ox -DIS_32 -DWANTVXDWRAPS -DVTOOLSD -
I$(INCLUDEVXD)
```

COPTFLAGS

COPTFLAGS is used to set the C/C++ optimizations. This can be used, for example, to disable optimizations during DEBUG builds.

DEVICETYPE

DEVICETYPE is used to select either a VxD (the default) or a Port Driver (.PDR) file.

DEVICETYPE = VX	Build a VxD (default)
DEVICETYPE = PORTDRIVER	Build a Port Driver

DEBUG

DEBUG allows the choice of linking with the VTOOLSD debug libraries. In addition, if **DEBUG** is set, the **DEBUG** symbol is defined when the source files are compiled, and debug information is included by the compiler and linker. The valid options are:

DEBUG = 0	Do not include debug information (default)
DEBUG = 1	Include debug information

DEVICENAME

DEVICENAME is used in the definition file read by the linker. This name must be the same as the argument used to the **Declare_Virtual_Device** macro in the source file so the linker can correctly identify the DDB to the VMM. The make system generates the definition file automatically with the extension .DEF. The **DEVICENAME** must be no more than 8 characters long. Example:

```
DEVICENAME = SIMPLE
```

DYNAMIC

DYNAMIC is used to control whether the driver will be dynamically or statically loaded. If **DYNAMIC** is set to 1, the DEF file is built with the **DYNAMIC** keyword, instructing the linker to set the appropriate flags to create a dynamically loadable driver. The valid options are:

DYNAMIC = 0	Build static driver (default)
DYNAMIC = 1	Build dynamic driver

FRAMEWORK

FRAMEWORK is used to select either the **C** or the **C++** VTOOLSD framework. This option controls which libraries are searched by the linker. The valid options are:

FRAMEWORK = C	Select the C framework
FRAMEWORK = CPP	Select the C++ framework

NDIS

NDIS is used to specify whether the Network Device Interface Specification (NDIS) libraries are searched by the linker. The valid options are:

NDIS = 0	Do not link with NDIS libraries (default)
NDIS = 1	Link with NDIS libraries

OBJECTS

OBJECTS is used to specify the list of object files used to build the driver. The list can contain one or more object file names, separated by one or more spaces. The file extension (e.g. **.OBJ**) must be specified. For example:

OBJECTS = SIMPLE.OBJ

OBJECTS = COMPLEX.OBJ UTILITIES.OBJ

OBJPATH

OBJPATH is used to specify the path used to locate and store the object files specified in the **OBJECTS** list. If you prefer maintaining object files in a directory other than the current directoy, set the **OBJPATH** symbol. For example:

SOURCEPATH = src

OBJPATH = obj

OBJECTS = obj\logger.obj obj\loginit.obj

SOURCEPATH

SOURCEPATH is used to specify the path used to find source files specified by the **OBJECTS** list. If you prefer maintaining source files in a directory other than the current directory, set the **SOURCEPATH** symbol. See the example in the **OBJPATH** description.

TARGET

TARGET specifies the oldest operating system on which the VxD is to run. Since each version of the O/S adds new services, it is necessary to screen out prototypes of some services from the header files when an older O/S is selected.

USER_LIB

USER_LIB is used to designate one or more user libraries to be searched during the link. For Microsoft builds, multiple libraries should be separated by spaces. For Borland builds, multiple library names should be separated by plus signs. For example:

USER_LIB = mylib1.lib mylib2.lib (Microsoft)

USER_LIB = mylib1.lib + mylib2.lib (Borland)

Do not use pathnames with embedded spaces. Use the short filename instead.

XAFLAGS

XAFLAGS is used to define additional arguments to the assembler. These arguments will be passed in addition to the standard arguments provided by the VtoolsD make system.

XFLAGS

XFLAGS is used to add additional options to the compiler command line. For example, this option might be used to modify the default optimizations selected by the VTOOLSD build system.

XLFLAGS

XFLAGS is used to add additional options to the linker command line.

NOFILTER

NOFILTER might be defined to suppress filtering of extraneous LINK and EDITBIN messages. If NOFILTER is *not* defined (as in the default build), then certain error messages will be hidden by the VTOOLSD FLTRWARN program. For example:

```
NOFILTER = 1 (default = undefined)
```

USER_COMMENT

USER_COMMENT can be set to define a comment for the DEF file. If no **USER_COMMENT** is specified, then a default comment is used. For example:

```
USER_COMMENT = Simple Test VxD
```

Object Files and Segment Selection Using Borland C/C++

The VxD loader allows different types of segments, as described in Chapter 3. Chapter 8 describes the segmentation scheme in greater detail. Here are the full details of the options available:

The Borland C/C++ compiler supports command line switches that allow the developer to control the emitted names and classes of the code and data segments. You can also use preprocessor statements inside a code module to change the current code segment, provided there is no in-line assembler code in the module. However, the make system relies on the command line switches for segmentation control.

If you are using the Borland compiler, then, by default, VTOOLSD compiles source modules with the same type of code and data segment. For example, the code and data segments for a module can both be LOCKED, or can both be INIT. It is possible, however, to compile a module with different code and data segment types. If, for example, a module requires code segment type PAGEABLE and data segment type LOCKED, you must make it a LOCKED module and control the code segment type with the preprocessor.

In order to determine the desired segment type for a given module, the VTOOLSD make system relies on the file extensions of the object files you provide in the **OBJECTS=** statement of the make file. For each type of segment, there corresponds a unique file extension, as shown by the following table:

Segment Type	Object File Extension
LOCKED	.obl, .obj
INIT	.obi
PAGEABLE	.obp
STATIC	.obs
DEBUGONLY	.obd

To cause the code and data segments of a given module to be a particular type, set the file extension of the object file name you provide in the OBJECTS= statement of the make file to the extension in the above table that corresponds to the desired segment type.

Here is an example to illustrate the usage of object file extensions:

```
OBJECTS = main.obl  initdev.obi  mod.obp
```

If the above line appears in a make file, then the VTOOLSD make system will compile module **main.c** (or **main.cpp**) with LOCKED segments, module **initdev.c** (or **initdev.cpp**) with INIT segments, and module **mod.c** (or **mod.cpp**) with PAGEABLE segments.

The VTOOLSD Make Files

VTOOLSD includes subsidiary make files used to define and implement the various rules and options required to build a VxD. The three files described here (USER.MAK, VTOOLSD.MAK, and VXDTARG.MAK) must be located in the VTOOLSD INCLUDE subdirectory.

USER.MAK

USER.MAK contains build information specific to your installation. This includes paths for certain tools and any changes you make to the default compile and link command line switches. For more information about the file paths contained in **USER.MAK**, please refer to Chapter 2.

USER.MAK contains paths used to locate the compiler, linker, and other tools necessary for building a VxD. The SETUP program creates this file interactively. If the path to a required tool changes, you should edit **USER.MAK** to update the path definitions.

You can also modify the behavior of VTOOLSD by redefining the compiler and linker switches in **USER.MAK**. Examine **VTOOLSD.MAK** to see which switches VTOOLSD uses by default.

The Compiler

The current VTOOLSD libraries are built to work the Microsoft Visual C/C++ compiler and the Borland C/C++ compiler. Versions supported at this writing are shown in this table:

Vendor	*COMPILER*=(in USER.MAK)	Version
Microsoft	MS5	Visual C++ 5.0
	MS5 or MS6	Visual C++ 6.0
Borland	BCB3	Borland C++ Builder

Compuware provides header files that define all of the services available for use in VxDs. You should not include any of the header files from the compiler's INCLUDE subdirectory. The VTOOLSD make system does not add this directory to the include search path.

You can use in-line assembly code in VxDs that you write. If you use in-line assembly with the Borland compiler, you must first install TASM32.

The Linker

TLINK32.EXE comes with Borland C/C++ 4.x. TLINK32 produces .DLL files in the PE format, which must be converted to VxDs by the utility, PELE. PELE is described below. Compuware has found the .DLL files produced by the Borland TLINK32 are sometimes corrupt. One observable symptom is that sometimes the first two bytes of the file are not "MZ" as they must be for a valid executable. You might be able to avoid this problem by splitting your driver source into fewer source files. You can also contact Borland for information about a possible fix or upgrade.

Using the Microsoft Visual C++ Developer Studio Environment

Visual C++ users can simply open the .MAK file of a VxD project as an external makefile, as long as environment variable VTOOLSD is defined. The integrated environment allows you to edit your source code and invoke **nmake** with a button click. Detailed instructions are provided in the online help file.

The SEGALIAS Utility

SEGALIAS is a utility that modifies OMF object files to conform to the segment conventions required to build consistent and correct VxDs. This utility is installed as part of the VTOOLSD toolkit.

SEGALIAS takes input from a line-oriented file. Each line specifies a segment name change. Segments are identified by a name and a class. A full change specification consists of an old-name/old-class pair and a new-name/new-class pair.

In addition to changing segment names, SEGALIAS prepares an object file for linking into a DLL in PE format which will be converted to a VxD by the PELE utility.

Command Line Syntax

Here is the syntax for invoking SEGALIAS:

```
C> segalias [options] command-file object-file
```

object-filename stands for the name of the OMF object file to be modified. command-file stands for the name of the auxiliary input file that contains lines that specify segment name changes.

Command Line Options

The following table summarizes the command line options for PELE. Further information is provided below:

Option	Meaning
-p	Prepare the object file for linking into a DLL in PE format for later processing by PELE.

All OMF object modules that comprise a VxD built using PELE should be processed by SEGALIAS with the -p switch. This includes all object modules produced by the Borland compiler or TASM32. Specifying the -p switch adds information to the object file which would otherwise be lost during the link phase, but which is necessary to generate a correct VxD.

Syntax of the Auxiliary Command File

The command file for SEGALIAS is a line-oriented file that specifies the name changes to perform on the input OMF object file. Lines have the following syntax:

```
S old-name old-class new-name new-class
```

where old-name and old-class specify the segment's name and class as found in the input object file, and new-name and new-class specify how that segment should appear after processing by SEGALIAS.

Here is an example input line from the SEGALIAS command file:

```
S _TEXT CODE _LTEXT LCODE
```

The above line directs SEGALIAS to change the object file so the segment named _TEXT whose class is CODE is renamed _LTEXT with class LCODE.

PELE: The PE to LE Conversion Program

PELE is a tool that converts DLLs from the PE (Portable Executable) format to LE (Linear Executable) format, the latter being the format of VxDs. This tool enables you to create VxDs with a language system whose the native linker does not directly support that format. The VTOOLSD make system automatically invokes PELE with the correct command line parameters, so it is unlikely you will need to familiarize yourself with its operation.

Command Line Syntax

Here is the syntax for invoking PELE:

```
C> pele [options] input-filename
```

`input-filename` stands for the name of the DLL (in PE format) you are converting to a VxD. If you specify an output file with the -o option, PELE does not modify the input DLL.

Command Line Options

The following table summarizes the command line options for PELE. Further information is provided below:

Option	Meaning
-k#	Specifies the DDK version to be set in the VxD header
-o <filename>	Specifies the name of the output VxD
-r <filename>	Specifies the name of the .COM file containing the real mode initialization code for the VxD
-m <filename>	Specifies the name of the map file
-c <filename>	Specifies the name of the auxiliary command file
-v and -V	Enables verbose output to stdout. For use by Compuware support personnel

All file names must be fully specified; PELE does not support any default extensions.

Specifying the DDK Version. The -k option allows you to specify the DDK version to be stamped on the VxD header. The argument for this option is one of the following numbers, specified in hexadecimal:

- 30A for Windows 3.10
- 30B for Windows 3.11
- 400 for Windows 95

For Example:`C> pele -k400 YN6000.DLL`

Specifying the Output File. The **-o** option allows you to name the output VxD. If you omit this option, PELE modifies the input DLL. When you use this option, PELE writes the VxD to the specified file and does not modify the input DLL. If the specified file already exists, PELE overwrites it.

For Example:
```
C> pele -o YN6000.VXD YN6000.DLL
```

Specifying the Map File. While you are debugging your VxD, you typically need symbol information found in a map file. Because PELE changes the ordering of segments in the conversion process, the map file produced by the linker does not reflect the actual symbol addresses in the output VxD. For this reason, you must provide PELE with the name of the map file so it can update the symbol addresses. PELE uses the same map file for input and output. After processing by PELE, the map file can be used as input to 's MSYM or Microsoft's WDEB386 to produce a valid symbol file for your debugger.

For Example:
```
C> pele -o YN6000.VXD -m YN6000.MAP YN6000.DLL
```

Specifying the Real Mode Initialization File. If your VxD requires a real mode initialization segment, you must build a .COM file containing the initialization code and pass its name to PELE. The **-r** option allows you to specify the name of this .COM file, which PELE inserts into the output VxD as the real mode initialization segment.

The entry point of the .COM file becomes the real mode initialization entry point of the VxD. Windows calls this entry point when the VxD is loaded, prior to switching to protected mode.

Code in a .COM file is contained within a single segment, and the entry point must be at offset 0x100 in that segment. In assembly language, use the ORG directive to set the code origin immediately after the segment declaration. The label at this location should also be specified as a parameter to the END directive in order to mark it as the file entry point.

When a VxD is dynamically loaded, the real mode initialization section is ignored.

For Example:
```
C> pele -o YN6000.VXD -r YN6RINI.COM YN6000.DLL
```

Specifying the Auxiliary Command File. The -c option allows you to specify the name of the auxiliary command file. The command file contains additional commands that control how PELE converts the DLL to a VxD. The syntax of the command file is described later in this section.

For Example:
```
C> pele -o YN6000.VXD -c YN6000.PEL YN6000.DLL
```

Syntax of the Auxiliary Command File

Each line of the command file contains one directive for PELE. The following table summarizes the directives and their purpose:

Directive	Meaning
`DYNAMIC`	Specifies the output VxD is to be marked as dynamically loadable

MERGE <section-list>	Specifies the sections named in the following list are to be merged in the output VxD
ATTRIB <section> <attribute>	Specifies an attribute for a section

DYNAMIC Directive. If the directive DYNAMIC appears in the command file, PELE sets flags in the VxD header indicating the VxD can be dynamically loaded. Windows 3.10 does not load VxDs dynamically.

MERGE Directive. The MERGE directive specifies a list of sections in the input DLL that are to be combined into a single segment in the output VxD. Sections merged together inherit the attributes of the leftmost named section in the list. To determine the names of the sections in a DLL, look at the map file or use a binary dump program.

For Example:`MERGE LCODE BSS TLS CONST`

ATTRIB Directive. The ATTRIB directive instructs PELE to assign a specific attribute to the segment in the output VxD that corresponds to a specified section in the input DLL. The keyword ATTRIB must be followed by a section name (which can be obtained from the map file) and an attribute keyword. The attribute keywords are:

LOCKED	The segment in the output VxD is locked
INIT	The segment in the output VxD is discarded after initialization
STATIC	The segment (in a dynamically loaded VxD) remains loaded after the VxD is unloaded
PAGEABLE	The segment in the output VxD is pageable
DEBUG	The segment in the output VxD is loaded only when a debugger is present

For Example:`ATTRIB LCODE LOCKED`

Loading a VxD

Loading Static VxDs

Static VxDs are loaded by the Virtual Machine Manager when Windows starts up. To request the VMM load your VxD, insert a line into the **SYSTEM.INI** file specifying the path of your VxD. The **SYSTEM.INI** file is located in the Windows directory. For example, to load the **SIMPLE.VXD** VxD from the VTOOLSD C example directory, add the following line in the [386Enh] section of **SYSTEM.INI**:

```
device=c:\vtoolsd\examples\c\simple\simple.vxd
```

When you next start Windows, your VxD will be loaded. The order of the lines in **SYSTEM.INI** is not critical; VxDs are initialized according to the INIT_ORDER specified in the Device Descriptor Block.

You can add additional lines to **SYSTEM.INI** with parameters that control aspects of your VxDs behavior. These lines can be read at initialization time -- that is, when your VxD is processing one of the INIT control messages. There are a set of VMM functions VxDs use to read information from **SYSTEM.INI**. These services are available only at initialization time

By convention, you should create a new section in **SYSTEM.INI** that corresponds to the name of your VxD. For example, the LOGGER example checks for a parameter LOGSIZE in the [LOGGER] section, and, if found, uses the value of the parameter to set the initial size of the log buffer. The following lines are added to **SYSTEM.INI** to specify a 2K buffer:

```
[Logger]

Logsize=2048
```

If the VMM is unable to load a VxD (if the file is not found, or is not formatted correctly) the system will pause and display a message.

Loading VxDs Dynamically

Under Windows 95, VxDs can be loaded dynamically. To build a VxD marked for dynamic loading, simply include the line "**DYNAMIC=1**" in the make file.

To test your dynamically loadable VxD, you can use the Driver Monitor utility provided with VtoolsD. You can start Driver Monitor from the VtoolsD submenu. To load your driver, simply select **File | Open Driver** and choose the driver from the ensuing file dialog. Once the driver is loaded, you can start it by selecting **File | Start Driver**. Driver Monitor makes the required call to the Service Control Manager to start your opened driver.

Once loaded, your VxD will receive control message SYS_DYNAMIC_DEVICE_INIT. This is the only initialization message it will receive.

To test unloading your driver, simply choose **File | Stop** from the Driver Monitor menu. Note that if your driver does not include and Unload function, the Service Control Manager cannot unload it. Also note that the system maintains a reference count that tracks the number of times the driver was started and stopped. The driver will not be stopped (unloaded) unless the number of stop requests equals the number of start requests.

Loading with the Registry

Windows 95/98 allows you to cause a static VxD to be loaded by adding an entry to the system Registry. The registry entry to add your device driver to is HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VxD. You can use **REGEDIT.EXE** (supplied with Windows 95) to make this modification. First, add a subkey whose name is the device name of your VxD (e.g. XYZDEV). Then, add a string value to the new subkey whose name is "StaticVxD" and whose data is the full pathname of your VxD file.

Loading I/O Subsystem Drivers

The I/O Supervisor of Windows 95/98 loads dynamically loadable VxDs found in directory **\WINDOWS\SYSTEM\IOSUBSYS** when Windows starts up. This includes SCSI miniport drivers with extension .MPD. This feature is only to be used by drivers that register with the IOS.

Debugging

VxDs are notoriously difficult to debug. There are several techniques that make the job somewhat easier.

Use the Debug Kernel

The debugging version of the operating system is provided with the Microsoft DDK.

Test Code Outside of the VxD Environment

Whenever possible, test modules, routines, and even code fragments outside of the VxD environment. You can test 32-bit code in a DOS or Windows environment in several ways:

- Use Win32s to test 32-bit code within a Windows application.
- Test 32-bit code using Windows or Console applications running on Windows NT or Windows 95.

Many routines can be written for a 16-bit environment as well, and then ported into the 32-bit VxD environment after you have tested and debugged them as thoroughly as possible.

Add Error Checking and Validation Code

Liberally sprinkle your VxD with error-checking and parameter validation code. You can enclose code within a DEBUG conditional to exclude extra checking from your retail product. If the DEBUG variable is defined in the environment, in the make file, or on the NMAKE command line, the DEBUG symbol will also be defined when the VxD is compiled:

```
#ifdef DEBUG

    if (param == suspicious_value)

    {

    DEBUGWARN("Warning: param has a suspicious value");

    }

#endif
```

Alternately, you can use the ASSERT macro defined in the VTOOLSD header files:

```
Assert(param != suspicious_value);
```

The ASSERT macro only generates test code if DEBUG is defined. If the assertion fails, a message is printed to the current debug output device and a breakpoint interrupt is issued to enter the debugger.

Debug Monitor

Compuware provides a utility which can display VxD debug messages in a Windows application. The **dprintf** or **nprintf** services, similar to the familiar **printf** service in the C run-time library, can be used to generate messages to be displayed by the debug monitor.

VxD Debuggers

You can use two debuggers for VxD testing and debugging. Compuware's SoftICE is a powerful kernel debugger. The Wdeb386 debugger is supplied by Microsoft as part of the Windows Device Driver Kit (DDK). VTOOLSD is set up to allow you to choose which debugger to use and generates an appropriate symbol table file for the debugger of your choice.

Both of these debuggers work best when the debug kernel is installed. The debug version of the Windows kernel is available on the Windows DDK and provides additional error checking and diagnostic messages. Developing and testing with the debug kernel is highly recommended.

SoftICE

SoftICE supports source debugging and offers full-screen single machine debugging. SoftICE also supports output to a monochrome screen or use of a second computer connected through a serial link.

Wdeb386

Wdeb386 is designed to direct output to and receive commands from a second computer running a communications package over a serial link. Wdeb386 has a limited command set, cannot display source code, and does not support single-computer debugging. Debugging is strictly machine-level with disassembly provided. Wdeb386 is available as part of the DDK.

Additional Debug Support

The C Run-Time Library function dprintf provides a facility for generating trace and debug output which can be directed to a monochrome monitor or through a serial port to a second computer. Similarly, the Class Library provides a debug stream with similar capabilities.

The tutorial example for the class library (CLASSTUT.VXD) demonstrates the ability to redirect debugging output from your VxD to a Windows application window. The LOGGER example uses the C Framework to implement a log that can be used to queue messages asynchronously for later retrieval from a DOS or Windows program.

5 The VTOOLSD C Framework

Overview and Goals

The VTOOLSD C Framework is designed to make it easy to develop VxDs in C without resorting to assembly language. The C Framework provides macros and library routines that are used to define the Device Descriptor Block and create handlers for all events to which a VxD might need to respond. Default handlers are provided as needed.

This chapter begins with an annotated listing describing in detail how the C Framework is used to create a simple VxD. The QuickVxD documentation in *Chapter 3: QuickVxD* on page 35 provides additional information about some of the contents of these files.

An Annotated Example

The QuickVxD program can generate the initial skeleton of your VxD. The files listed below (SIMPLE.C and SIMPLE.H) were generated by QuickVxD with the following settings:

Device Name	SIMPLE
Device ID	`UNDEFINED_DEVICE_ID`
Device Init Order	`UNDEFINED_INIT_ORDER`
Device Major Version	`1`
Device Minor Version	`0`
Entry Points	`none checked`
Framework	`C`
Build	`Debug`
Service Prototypes	`none added`
Control Messages	`DEVICE_INIT checked`

The SIMPLE VxD

The SIMPLE VxD, which can be found in the **EXAMPLES\C\SIMPLE** subdirectory, demonstrates a VxD that does essentially nothing but load and harmlessly monitor control messages. First, we present complete listings for the source and header files that constitute the project. Following the listings is a detailed annotation that describes how the VxD is constructed, with an emphasis on the Device Descriptor Block definition and the Control Handler routines.

SIMPLE.H

// **SIMPLE.H** - include file for VxD SIMPLE

```
#include <vtoolsc.h>

#define SIMPLE_Major 1

#define SIMPLE_Minor 0

#define SIMPLE_DeviceID UNDEFINED_DEVICE_ID

#define SIMPLE_Init_Order UNDEFINED_INIT_ORDER
```

SIMPLE.C

// **SIMPLE.C** - main module for VxD SIMPLE

```
#define   DEVICE_MAIN

#include  "simple.h"

#undef    DEVICE_MAIN

Declare_Virtual_Device(SIMPLE)

DefineControlHandler(DEVICE_INIT, OnDeviceInit);

BOOL ControlDispatcher(

    DWORD dwControlMessage,

    PCLIENT_STRUCT pClientRegs,

    DWORD EBX,

    DWORD EDX,

    DWORD ESI,

    DWORD EDI)

{

    START_CONTROL_DISPATCH
```

```
        ON_DEVICE_INIT(OnDeviceInit);
    END_CONTROL_DISPATCH
    return TRUE;
}
BOOL OnDeviceInit(VMHANDLE hVM, PCHAR CommandTail,
    PCLIENT_STRUCT pClientRegs)
{
    return TRUE;
}
```

SIMPLE.H Annotated

SIMPLE.H is designed to be included by any source files used to build the SIMPLE VxD. In turn, **SIMPLE.H** includes **VTOOLSC.H**, the C Framework header file that resides in the VTOOLSD **INCLUDE** subdirectory.

```
#include <vtoolsc.h>
```

Several constants are defined. These constants are used when the Device Descriptor Block is constructed from the declaration in SIMPLE.C.

```
#define SIMPLE_Major1
#define SIMPLE_Minor0
#define SIMPLE_DeviceIDUNDEFINED_DEVICE_ID
#define SIMPLE_Init_OrderUNDEFINED_INIT_ORDER
```

SIMPLE.C Annotated

SIMPLE.C uses the C Framework to create a very simple VxD. This VxD includes a handler for the **DEVICE_INIT** control message, but does no processing.

The symbol **DEVICE_MAIN** is defined before including the header file. The purpose of this symbol will be described later, under the heading of "Providing VxD Services."

```
#define   DEVICE_MAIN
#include  "simple.h"
#undef    DEVICE_MAIN
```

The **Declare_Virtual_Device** macro creates the Device Descriptor Block (DDB). The DDB holds the data required by the Virtual Machine Manager to call services exported by the VxD. The macro takes the device name as an argument; it must be eight characters or fewer. The C Framework pads the device name in the DDB with trailing blanks to make it exactly eight characters.

The name should *not* follow the Microsoft convention of naming devices starting with a 'V' and ending with a 'D'. Instead, choose a name that is likely to be unique; perhaps by embedding your company or product name.

```
Declare_Virtual_Device(SIMPLE)
```

The **DefineControlHandler** macro declares a handler for a control message. The macro generates a function prototype that allows the compiler to perform argument type checking against the handler function when it is defined.

```
DefineControlHandler(DEVICE_INIT, OnDeviceInit);
```

The **ControlDispatcher** routine is called by the C Framework whenever a control message is available. The **ControlDispatcher** routine is called with a set of arguments that were passed in registers to the C Framework control message handler for the VxD. The C Framework translates the Boolean return value from the control handler to appropriate flag settings for the particular control message.

The **START_CONTROL_DISPATCH** macro initiates a **switch** statement that selects a control message. The **ON_xxx** macros generate CASE statements that call the appropriate handler with the correct arguments selected from the register values passed into the **ControlDispatcher** routine.

You should not need to modify the **ControlDispatcher** routine, aside from possibly adding additional message dispatches using the **ON_xxx** macros. These macros are defined in the **VTOOLSC.H**.

```
BOOL ControlDispatcher(
    DWORD dwControlMessage,
    PCLIENT_STRUCT pClientRegs,
    DWORD EBX,
    DWORD EDX,
    DWORD ESI,
    DWORD EDI,
    DWORD ECX)
{
    START_CONTROL_DISPATCH
        ON_DEVICE_INIT(OnDeviceInit);
```

```
    END_CONTROL_DISPATCH

    return TRUE;

}
```

The **OnDeviceInit** routine is called when the **DEVICE_INIT** control message is detected. Most VxDs will do required initialization within this handler. This simple VxD has no initialization, and so returns **TRUE** to indicate no errors occurred and Windows should continue loading.

```
BOOL OnDeviceInit(VMHANDLE hVM, PCHAR CommandTail,

          PCLIENT_STRUCT pClientRegs)

{

    return TRUE;

}
```

Control Message Handling

Control message handlers can easily be implemented in C. The **ControlDispatcher** routine, as described in the SIMPLE.C annotation section above, ensures that arguments are passed correctly to a VxD's control message handlers. Unhandled messages are simply ignored, and a success code passed back to the Virtual Machine Manager when necessary.

Details on the meaning and arguments to specific control messages can be found in the on-line help files.

Providing Services

Many VxDs are designed to provide services to other programs. VxDs can provide services to any program running in the Windows environment. Different techniques are used to create interfaces to other VxDs, Windows programs, DOS programs, and DOS Extended protected mode programs.

Providing Services to Other VxDs

VxDs can export services that can be called by other VxDs. The C Framework can create the tables required to register such services with the VMM.

Most VxDs will not need to define services called by other VxDs. If your VxD needs to do so, you should contact Microsoft to obtain a unique VxD ID other VxDs can use to identify your services.

If VxD services are exported, the first service (service # 0) must be a **Get_Version** service. VxDs wishing to call services defined by your VxD will call **vxdname_Get_Version** first, where **vxdname** is the device name of your VxD. The VMM will fail this call without generating a fault, even if your VxD has not been loaded. Calls to anything other than service 0 will cause a fault if the target VxD is not loaded.

The C Framework defines several macros that are used to create the VxD service table in the header file. When this file is included in the main module of your VxD, the symbol **DEVICE_MAIN** should be defined. This causes the actual dispatch table to be generated. If the header file is included in another VxD that calls the services described in the VxD Service Table, the symbol **DEVICE_MAIN** must be left undefined. This results in a set of definitions that allow the **VxDCall** macro to work correctly when the services are called.

For example, we might extend **SIMPLE.H** to define a **Get_Version** service. The service is prototyped in the header file:

```
DWORD SIMPLE_Get_Version();
```

The services are defined using the Service Table macros:

```
Begin_VxD_Service_Table(SIMPLE)

    VxD_Service(SIMPLE_Get_Version)

End_VxD_Service_Table
```

The implementation of the service appears in **SIMPLE.C**:

```
DWORD SIMPLE_Get_Version()

{

    return (SIMPLE_Major << 8) | SIMPLE_Minor;

}
```

Because the **Get_Version** service should always be defined if the VxD provides any VxD services, QuickVxD generates a default **Get_Version** service entry in the table and provides a default implementation of the service. Remember the Device ID must not be **UNDEFINED_DEVICE_ID** if VxD services are exported.

The code described above is generated automatically by QuickVxD when the user positions the cursor in the **[C Prototypes for Exported VxD Services]** box and clicks the **[Add]** button. You can add additional services by typing in the function prototypes and adding them in the same manner.

Providing Services to V86 and Protected Mode Programs

Although it is rarely required since the early days of Windows 3.1, VxDs can also provide services to Virtual 8086 (DOS) programs and Protected Mode (DPMI or Windows) programs. QuickVxD creates handler routines if you specify the [Virtual 8086 Mode] or [Protect Mode] Entry Point options.

If you define a routine with the name **V86_Api_Handler**, the C Framework will call your routine when a V86 client program calls the V86 entry point provided by the VMM. Similarly, if you define a routine with the name **PM_Api_Handler**, the C Framework will call your routine when a Protected Mode program calls the PM entry point provided by the VMM.

Both the V86 and PM handlers receive the same arguments, which consist of the handle of the current Virtual Machine and a pointer to a Client Register Structure containing the register values of the application that is calling the VxD.

Obtaining a VxD Entry Point From an Application

Under Windows 3.x, a VxD was required to have a unique device ID in order for its PM or V86 application entry points to be accessible from applications. Windows 95 added support for a name based method to obtain a VxD entry point.

To obtain the VxD entry point using a unique device ID, you must:

- Load the AX register with 0x1684
- Load the BX register with the device ID of the VxD whose entry point is sought

To obtain the VxD entry point using a device name, you must:

Windows 95

- Load the AX register with 0x1684
- Set the BX register to zero
- Set ES:eDI to point to the eight character, space padded, uppercase, ASCII device name of the VxD whose entry point is sought
- Issue INT 0x2F

If the VxD is found, the application can invoke the VxD's API handler by issuing a far call to the address the INT 0x2F returns in ES:eDI. If the VxD is not found, ES:eDI is returned as a null pointer.

32-bit Windows applications cannot use this mechanism. Instead, they call into a VxD by first obtaining a file handle, and then invoking the Win32 API **DeviceIoControl**. See Chapter 9 for a complete description of this mechanism.

Providing Services Using Vendor Specific Entry Points

VTOOLSD provides an alternate way for VxDs to implement services that can be invoked from DOS, Windows, or DPMI clients without requiring a VxD ID allocated from Microsoft. The addition of support for name-based VxD calling in Windows 95 makes this mechanism obsolete. Although it is still supported, it is *not* recommended unless source compatibility with a Windows 3.x VxD is required.

In the Vendor Specific Entry Point scheme, the entry points are located dynamically by the client program using a unique ASCII string that you, the VxD writer, define. The string, referred to as the "Vendor Specific String", can be of any length, and is a case-sensitive string of characters terminated by a zero byte.

When a client seeks to call a vendor specific service, the client program issues an INT 2Fh with AX set to 168Ah and DS:SI set to point to the desired Vendor Specific String. The VxD that recognizes the requested string returns (via a routine supplied by the C Framework) an address the client can call in ES:DI.

When the client calls the address provided, control is transferred to the VxD by the VTOOLSD Framework.

For a working example, see the **V86VEND.C** in the VTOOLSD **EXAMPLES\C\V86VEND** directory. A sample DOS client program is also provided in the DOS subdirectory of the V86VEND example.

QuickVxD can be used to handle the details when creating the VxD. Select the **[Vendor Specific V86 Mode]** or **[Vendor Specific Protect Mode] entry point**. You must also specify a unique string that will identify your VxD. We recommend using a combination of your company name and product name, or any other string you expect will be unique.

Handling Events

Thunks

VxDs can install handlers for many different events that occur during the normal or abnormal operation of Windows. When calling a service to request an event notification, a VxD supplies the address of a callback procedure to be called when the event occurs.

In most cases, Windows passes arguments to the callback procedure in registers, which creates a problem if the callback procedure is to be written in C. To correctly invoke a callback procedure written in C, the processor must execute a few instructions to push the appropriate registers onto the stack. This short patch of code that binds the system's register interface to the C callback is called a **thunk**.

VTOOLSD makes it very easy to use C callback procedures. For each VMM/VxD service that uses a callback procedure requiring a thunk, there is a corresponding typedef of a thunk for that service in **INCLUDE\VTHUNKS.H**. Your VxD is only responsible for allocating the correct type of thunk and passing its address to the wrapper function. The VTOOLSD library automatically initializes the thunk with the correct code and directs the system to invoke the code in the thunk when the event in question occurs.

Allocating Memory for Thunks

Thunks can be statically allocated in the locked code segment, allocated from the locked heap using the **HeapAllocate**() service, or, rarely, allocated automatically on the stack (if you are certain they will not be used after the current frame is freed.)

Thunks have several characteristics important for the C programmer to remember:

- Thunk blocks must generally be in locked memory.
- Thunk blocks are small (typically 10-40 bytes) and fast.
- Routines in the VTOOLSD libraries store the callback address inside the thunk. Thunks can be used simultaneously for multiple events, provided they have the same callback address. Thunks can be targeted to a different callback only when it is no longer possible that the VMM or other VxD will invoke the callback for which the thunk was originally allocated.

Implementation Details

At least two arguments are required when registering a callback: the address of a callback handler and the address of a memory block where the thunk can be safely stored.

When a VxD calls a service that requires a callback, VTOOLSD copies a short code fragment into the thunk memory provided by the caller and patches the code with the address of the caller's callback. The code fragment used is responsible for converting any register and flag arguments from the VMM to corresponding stack-based arguments to the C callback. In situations where callbacks return a value, the thunk code also translates the return value to the appropriate register and flag arguments before returning to the VMM.

IMPORTANT: Most callbacks are defined as __stdcall. This means the callback pops its arguments off the stack upon returning. Failure to correctly declare your callback as __stdcall results in the callback crashing when it returns. Take care to match the definition of your callback with the appropriate typedef in INCLUDE\VTHUNKS.H. If the compiler issues a warning about the callback address argument passed to a VMM service, resist the urge to cast the address unless you are confident you understand the warning.

Thunk Example

Here is an example of a code fragment using a thunk for a callback.

```
Idle_THUNK  thunkIdle;      // Allocate memory for the thunk

// This is the C routine that will be called on idle:
BOOL __stdcall MyIdleHandler(VMHANDLE hVM, PCLIENT_STRUCT pcrs)
{
    // Do any idle processing here
    return FALSE;           // TRUE if we take a long time to run
}

// Install the handler, perhaps during a control message
Call_When_Idle(MyIdleHandler, &thunkIdle);
```

Avoid a Common Mistake

Many services that have callbacks require the address of thunk. A common mistake is to declare a variable of type "pointer to thunk" and pass it uninitialized to the service. The correct operation is to declare a variable of type "thunk" and pass its address to the service.

Writing Callbacks in Assembler

If you determine you must write your callback in assembler, you can still call the service that arranges the callback from C or C++. VTOOLSD provides a way to mix register-based and C-based callbacks. If a thunk address of NULL is passed as an argument to a service that requires a callback, then the system assumes the callback is already a register-based procedure and passes the calling VxD's callback address to the VMM or VxD service.

6 The VTOOLSD Class Library

Introduction

The VTOOLSD class library provides an object-oriented framework for developing VxDs in C++. The primary goal of the library is to reduce the time required to develop robust and efficient VxDs. In many software projects, a fraction of the total development time is devoted to learning the details of the programming environment and the remainder to actually developing code specific to the project. In development of VxDs, a large fraction of the development time is often spent dealing with the complexity of the VMM and other system VxDs. The class library addresses this issue in several ways:

First, the C++ language itself has features that enhance programmer productivity. Strict type checking and type-safe linking eliminate many bugs before execution. The use of classes to encapsulate code and data results in better structured VxDs. Reuse of classes within a single VxD, or across multiple projects, eliminates the need to reinvent programming constructs in each new context where the need might arise.

The class library expedites development by providing a framework on which to build VxDs. To add a VxD service, dispatch a control message, or provide an application interface, you simply add or override a member function in the class that serves to abstract your VxD. This structure is familiar to programmers who have used object-oriented application frameworks in other event-driven programming environments.

The classes that comprise the library are designed to present a higher level interface than that presented by the VMM. By combining operations commonly used together, the class library simplifies VxD programming. For example, consider the operation of hooking a protected mode interrupt. The assembly language or C programmer must first figure out it is necessary to allocate a protected mode callback, and must then use a seemingly unrelated service to set the protected mode interrupt vector. The class library presents the interrupt as a coherent object; member functions for hooking, handling, and unhooking are conceptually tied together within the class structure. In general, the classes allow the programmer to think in terms of objects that correspond to the high level functions of the VxD, as opposed to a vast set of fine-grained VMM services.

By presenting a consistent interface across a broad range of classes, the class library reduces the learning curve for VxD programming. Whereas the assembly language or C programmer must learn how to use a new set of VMM services each time he or she must trap a new kind of event, the C++ programmer discovers all classes used to subscribe to events have nearly identical member functions. This applies to various kinds of faults, interrupts, callbacks, I/O ports, hot keys, and other events.

While programmer productivity is a key concern, efficiency of the resulting code is of equal or greater importance. Because VxDs can seriously impact system performance, it is important always to keep in mind the efficiency of the code. Many programmers expect that C++ programs are big and slow, but this is not necessarily so. First of all, compiler optimization technology has improved greatly in recent years and usually yields very respectable code. Furthermore, key sections of the class library have been carefully written in assembly language. These sections include all the thunks that provide the linkage between the classes and the VMM. Finally, keep in mind that using the class library does not preclude you from using assembly language for the sections of your VxD most critical to performance. It is easy to link in an assembly language module, and even easier to use the compiler's in-line assembler capability when speed is paramount.

Two Class Libraries: Introducing Device Access Architecture

There are two class libraries in the current version of VTOOLSD.

The first is the original VTOOLSD class library, which contains both *framework classes* that provide a skeleton on which to build a VxD, as well as *utility classes*, which facilitate access to various operating system components.

The second library is the *Device Access Architecture* (DAA) library, which contains classes for accessing hardware and classes that abstract operating system components common to all Win32 platforms. The advantage of using this library is that the DAA classes are supported in the DriverWorks product.

Classes from the original VTOOLSD class library have names that start with 'V'. Classes in the DAA library have names that start with 'K'.

You can use the DAA library with the following settings in your makefile.

```
FRAMEWORK = CPP

TARGET = WIN95  or TARGET = WIN98
```

It is perfectly all right to use DAA classes and classes from the original class library in the same VxD. The extent to which you isolate usage of DAA classes in separate source modules will determine the degree of portability of your source code.

For an example of a driver that uses DAA, see *examples\cpp\daahwint*.

About this Chapter

To get the most out of this chapter, you should be familiar with the material presented in Chapter 3 and have a solid understanding of C++ concepts.

This chapter presents an overview of most of the classes in the original VTOOLSD class library. Although the tutorial involves working with 16-bit applications, most the discussion is pertinent to current Windows 9x platforms. The on-line documentation provides full descriptions of individual classes and member functions.

For a description of the DAA classes, see the online help, and the DriverWorks manual.

The Class Library: A Tutorial

The next several sections show the development of a VxD based on the VTOOLSD class library. The example VxD, **CLASSTUT.VXD**, works in conjunction with a helper application, **RELAY.EXE**, to display messages in a window whenever a new virtual machine is created or destroyed.

The VxD makes use of a utility program called **DBWIN.EXE**, found in the **BINW** subdirectory of the SDK. (You can also obtain **DBWIN.EXE** from the Microsoft Windows SDK Forum on CompuServe. It is included with Microsoft Visual C++ 16-bit edition.) DBWIN intercepts messages applications emit using the Windows API function **OutputDebugString**, and displays the messages in a window. Without DBWIN, **OutputDebugString** displays messages on a debugging terminal connected to the serial port.

The example VxD fields control messages the VMM issues when virtual machines are created or destroyed. It then sends a message to the helper application, **RELAY.EXE**, which in turn issues a call to **OutputDebugString**. DBWIN intercepts and displays this message.

Admittedly, this is a somewhat artificial example, but it illustrates several aspects of class library usage, including the following.

- How to use the framework classes as a starting point for a VxD.
- How to set up communication between a Windows application and a VxD.
- How to use the event processing classes of the class library.
- How to build on classes in the library to create a class that serves as a reusable software component.

You will find the source code for **CLASSTUT.VXD** in subdirectory **EXAMPLES\CPP\CLASSTUT** of your VTOOLSD installation.

The tutorial starts with the following section and continues through the section entitled “Interfacing to Applications”. The remainder of the chapter discusses additional classes from the library not required by the tutorial.

Getting Started with the Class Library

The first thing you must do is choose a name for your VxD. The name must begin with a letter be composed of up to eight upper case letters and digits. The convention of beginning VxD names with "V" and ending with "D" is for system VxDs supplied by Microsoft, so do not feel compelled to name your VxD in that way. In fact, Microsoft recommends you do *not* mimic this convention. Instead, choose a name likely to be unique, perhaps by including your company or product name. The name you choose becomes part of the Device Descriptor Block (DDB) for your VxD, and must uniquely identify your driver.

For the example VxD in the tutorial, we chose the name "CLASSTUT".

After choosing a device name, you define four constants. Normally, you place these definitions in the main include file for your VxD. The constants specify these pieces of information about your VxD.

- Device ID
- Initialization Order
- Major Version
- Minor Version

Device ID

The device ID is a 16-bit value that uniquely specifies your VxD. You need to define a device ID if your VxD provides callable services to other VxDs. If not, you can use the predefined value **UNDEFINED_DEVICE_ID**.

To set the device ID, define a constant of the form **xxx_DeviceID**, where xxx stands for the name you chose for your VxD. Our example VxD has the following line in the include file **CLASSTUT.H** to meet this requirement:

```
#define CLASSTUT_DeviceIDNUMEGA_TEST_ID
```

Do not use **NUMEGA_TEST_ID**. The value is reserved for Compuware.

Initialization Order

The second constant you must define specifies when, with respect to other VxDs, your driver receives initialization notifications. The initialization constant is an unsigned 32-bit value usually defined in terms of initialization constants for other devices. The higher the value, the later in the sequence of VxDs your driver is initialized.

You must define a constant of the form **xxx_Init_Order**, where xxx stands for the name you chose for your driver.

For example, suppose your driver requires initialization *after* the virtual display device, VDD, has been initialized. To ensure this, you express the initialization constant as follows.

```
#define MYVXD_Init_Order  V DD_INIT_ORDER + 1
```

The example VxD in this tutorial has no particular dependencies on other drivers, so we use the value **UNDEFINED_INIT_ORDER**. **CLASSTUT.H** includes the following line.

```
#define CLASSTUT_Init_Order_UNDEFINED_INIT_ORDER
```

Major Version and Minor Version

Finally, we need two constants that specify the major and minor version numbers of our driver. These two constants are of the form **xxx_Major** and **xxx_Minor**, where xxx stands for the device name.

We include these two lines in **CLASSTUT.H**.

```
#define CLASSTUT_Major1

#define CLASSTUT_Minor0
```

With these preliminaries out of the way, you can begin to build the framework of your VxD.

The Framework Classes

There are three classes in the class library, **VDevice**, **VVirtualMachine**, and **VThread** which play a special role in structuring the VxD.

The most important of the framework classes is **VDevice**. Each VxD that uses the VTOOLSD class library is built around a class derived from **VDevice**. The class library run-time system automatically creates a single instance of your device class when the VxD is loaded.

Instances of classes derived from **VVirtualMachine** correspond to virtual machines in the VMM environment. Many VxDs need to maintain data structures for each virtual machine in the system; this class provides a convenient means for doing so. Unlike **VDevice**, the VTOOLSD class library does not require you derive a class from **VVirtualMachine**, but in most cases you will want to do so.

The role of class **VVirtualMachine** with respect to virtual machines is exactly analogous to the role of class **VThread** with respect to threads. A VxD has the option of creating an instance of a class derived from **VThread** to correspond to each thread that exists in the system.

The member functions of the classes you derive from the framework classes determine the following characteristics of your VxD.

- What services your VxD supplies to other VxDs
- Whether or not your VxD supplies an entry point for applications
- Which control messages your VxD processes

Defining the Device Class

- To enable automatic construction of your device class, you must define the symbol **DEVICE_CLASS** in your VxD's include file. The value of the **DEVICE_CLASS** symbol is the name of the class you derive from **VDevice**.
- In the tutorial example, our device class name is **ClasstutDevice**. We add the following line to **CLASSTUT.H**:

```
#define DEVICE_CLASSClasstutDevice
```

Providing Services to Other VxDs

There are two simple steps to adding a C-callable service to your driver that other VxDs can invoke. First, you add a static member function to your device class, then you add the same function to your driver's service table.

In our tutorial example, we want to export one service, namely **CLASSTUT_GetVersion**. First, we add it to our device class as a static member function.

```
class ClasstutDevice : public VDevice

{

public:

    static DWORD CLASSTUT_GetVersion();
```

Next, we add the function to our driver's service table. The service table resides in the main include file for our driver, CLASSTUT.H. It begins by invoking the macro **Begin_VxD_Service_Table** and ends with the macro **End_VxD_Service_Table**. All intervening lines are invocations of the macro **VxD_Service**, which takes the function name as an argument.

The result for the example looks like this.

```
Begin_VxD_Service_Table(CLASSTUT)

   VxD_Service(CLASSTUT_Get_Version)

End_VxD_Service_Table
```

If your VxD supplies services to other VxDs, you should always have a *Get_Version* service as the first defined service. This will enable other VxDs to gracefully test for the presence of your driver.

Providing Entry Points for Applications

As described in Chapter 3, a VxD can supply entry points to both V86 applications and protected mode applications, including 16-bit Windows applications (see chapter 9 for information on 32-bit Windows applications). This enables applications to take advantage of the powerful capabilities of VxDs. Some virtual devices use this feature to provide a private API to a helper application. An example is the Virtual Display Driver, whose protected mode entry point is called by the enhanced mode grabber.

If you want to provide an entry point for protected mode applications, override member function **PM_API_Entry** in the class you derive from **VDevice**. Similarly, if you want to provide an entry point for V86 mode applications, override member function **V86_API_Entry** in your device class. You declare these functions within the declaration for your device class, as follows.

```
virtual VOID V86_API_Entry(VMHANDLE hVM, CLIENT_STRUCT* pRegs);

virtual VOID PM_API_Entry(VMHANDLE hVM, CLIENT_STRUCT* pRegs);
```

Our tutorial example does *not* require an application entry point, but if it did we would add one or both of the above lines to the declaration of class and then create a function along the following lines.

```
VOID ClasstutDevice::PM_API_Entry(VMHANDLE hVM, CLIENT_STRUCT*
pRegs)

{

    switch (_clientAX)

    {

    case 0x0001:

    case 0x0002:

        . . .

    }

}
```

A few notes about this function.

- Parameter *hVM* is the handle of the virtual machine that called the VxD. You can convert this to a pointer to the corresponding object derived from class **VVirtualMachine** by calling **VVirtualMachine::getVM**.
- Parameter *pRegs* is a pointer to a structure that contains the calling VM's register values. The VxD can read and write these values. Any changes the VxD makes are reflected to the calling VM when the application entry point returns. This is the most common means of passing arguments between an application and a VxD.
- Identifier *_clientAX* is a macro for `pRegs->CWRS.Client_AX`. This macro, along with those like it for all other registers, is defined in **VTOOLSCP.H**. Note you must use the name *pRegs* for the pointer to the register structure in order to use these macros. Branching on the passed value of AX is purely a convention and is not required.

Processing Control Messages

To this point, we have seen the framework classes, VDevice, VVirtualMachine, and VThread are used to define the services a VxD provides to other VxDs and to provide entry points for applications. The final important usage of these classes is to provide a structure for processing of control messages.

As described in Chapter 3, the VMM, or sometimes another VxD, broadcasts a control message by calling a dispatcher entry point in each VxD. The value in the EAX register at this entry point determines which control message is being sent.

When you use the VTOOLSD class library, member functions of your device class and the classes you derive from VVirtualMachine and VThread handle control messages. When you override a member function that corresponds to a control message, you implicitly install a handler for that message.

Using the Device Class to Process Control Messages

In our tutorial example, two of the control messages we want to process are DEVICE_INIT and CREATE_VM, so we override member functions **OnDeviceInit** and **OnCreateVM** in our device class. Our class definition now looks like this.

```
class ClasstutDevice : public VDevice

{

public:

        static DWORD CLASSTUT_GetVersion();

   virtual BOOL OnDeviceInit(VMHANDLE hVM, PCHAR szCmdTail);

   virtual BOOL OnCreateVM(VMHANDLE hVM);

};
```

With the class defined in this way, the VMM's broadcast of the DEVICE_INIT message will invoke our function **ClasstutDevice::OnDeviceInit**. Similarly, whenever the VMM broadcasts CREATE_VM, it will invoke our function **ClasstutDevice::OnCreateVM**. Dispatching of control messages to these handlers is done by the internals of the class library.

Using Virtual Machine Classes to Process Control Messages

Some control messages carry information specific to a particular virtual machine. Examples of such messages are VM_INIT, VM_SUSPEND, and QUERY_DESTROY. When you use the VTOOLSD class library, member functions of classes derived from **VVirtualMachine** are responsible for handling these control messages. Each such message identifies a virtual machine by its handle. The internal control dispatcher automatically locates the class instance associated with the specified virtual machine handle and invokes the appropriate member function.

You can recall there is a single instance of the device class, which the system automatically creates when the VxD is loaded. This is not true of the classes you derive from **VVirtualMachine**. You will be able to handle control messages associated with virtual machines only if you explicitly create instances of VM classes for those virtual machines. If there are virtual machines for which there is no instance of a class derived from **VVirtualMachine**, then the dispatcher will disregard any control messages targeted to those virtual machines.

In our tutorial example, we have just one class derived from **VVirtualMachine**, declared as follows.

```
class ClasstutVM : public VVirtualMachine
{
public:
   ClasstutVM(VMHANDLE hVM);
   VOID OnDestroyVM();
};
```

The first member function, a constructor, takes a virtual machine handle as an argument. The second member function is invoked by the internal control dispatcher whenever the VMM issues the control message DESTROY_VM. Note it has no parameters; the virtual machine in question is referenced by the implied "*this*" pointer. To obtain the virtual machine handle from an object based on VVirtualMachine, use function VVirtualMachine::getHandle.

What we need to do now is add code that constructs an instance of ClasstutVM. You will find the very simple constructor in CLASSTUT.CPP. The constructor simply passes its argument, the virtual machine handle, to the constructor for its base class, VVirtualMachine.

```
ClasstutVM::ClasstutVM(VMHANDLE hVM) : VVirtualMachine(hVM) {}
```

In the tutorial example, we invoke the constructor for the VM class in the handler for the control message CREATE_VM. The VMM broadcasts this message whenever a new virtual machine comes into existence. Notice the handler for CREATE_VM is a member function of the *device* class, not the VM class. At this stage, our handler for CREATE_VM looks like this.

```
BOOL ClasstutDevice::OnCreateVM(VMHANDLE hVM)
    {
            new ClasstutVM(hVM);
            return TRUE;
    }
```

When you allocate an instance of **ClasstutVM**, you invoke the constructor for **VVirtualMachine**. The constructor for **VVirtualMachine** stores the pointer returned by the invocation of **new** inside the Virtual Machine Control Block (VMCB) associated with the VM. The VMM maintains a VMCB for each virtual machine, and allows VxDs to allocate small amounts of private storage in it at initialization time. The initialization code for the class library reserves space in the VMCB for a pointer to the VM class instance corresponding to the virtual machine. The virtual machine handle is a pointer to the VMCB. This relationship makes it possible to efficiently dispatch control messages to member functions of class **VVirtualMachine**.

The tutorial example defines a single VM class and creates an instance of it for each virtual machine. The system is flexible enough to support more elaborate schemes. You could, for example, derive multiple classes from **VVirtualMachine** or a hierarchy of classes, with each class corresponding to a different type of VM from your VxD's perspective. Furthermore, you do not have to create instances of your VM class for every VM, nor do they necessarily have to be created in OnCreateVM. For example, you could choose to create instances only for virtual machines that invoke the application entry point. It is also quite reasonable to add data members to your VM classes in order to manage data on a per VM basis.

One thing you should *not* do, however, is create more than one instance of a VM class for a given VM, because only one of them can receive control message dispatches.

Using Thread Classes to Process Control Messages

Four control messages sent by the system are dispatched to thread objects, i.e., instances of classes derived from **VThread**. Your VxD must create these instances in order to receive the control messages directed to the corresponding thread. In general, class **VThread** behaves analogously to class **VVirtualMachine**. The control messages associated with threads are: THREAD_INIT, TERMINATE_THREAD, DESTROY_THREAD, and THREAD_NOT_EXECUTEABLE.

Our tutorial example does not process any of the messages associated with threads.

Declaring the Device Descriptor Block

For each VxD, the VMM maintains a data structure called the Device Descriptor Block, or DDB. VxDs built with VTOOLSD are no exception. The class library provides a macro named **Declare_Virtual_Device** you use to declare the DDB. It takes the device name as a parameter.

For our example, we add these two lines to CLASSTUT.CPP.

```
#define DEVICE_MAIN

.

.

.

Declare_Virtual_Device(CLASSTUT)
```

Since it modifies the definition of **Declare_Virtual_Device** and other macros, the definition of symbol **DEVICE_MAIN** must be made prior to the **#include** statements for the VTOOLSD include files. You must not define this symbol or use **Declare_Virtual_Device** in more than one of the source modules that comprise your VxD.

Before continuing, let us review where we stand with our example VxD.

- We have chosen a device name, CLASSTUT, and defined four constants that specify the device ID: initialization order, major version, and minor version.
- We have named our device class, **ClasstutDevice**, and added a **#define** line denoting it as such in the main include file.
- We have added a **GetVersion** service, callable by other VxDs, by declaring it as a static member function in our device class, and then adding a line to the driver's service table in the main include file.
- We have set up processing for control messages DEVICE_INIT and CREATE_VM by overriding virtual member functions in our device class.
- We have derived class **ClasstutVM** from **VVirtualMachine** and overridden one of its virtual member functions in order to process control message DESTROY_VM.
- We have added code to **ClasstutDevice::OnCreateVM** to create an instance of **ClasstutVM** each time a new virtual machine comes into existence.
- Finally, we have declared the Device Descriptor Block for our VxD using the macro **Declare_Virtual_Device**.

Debugging Classes

This section introduces some new classes for debugging you will find useful in the development of your VxD and that are relevant to the development of our tutorial.

Most C++ programmers are familiar with *iostreams*. They provide a powerful syntax for formatted input and output. The VTOOLSD class library provides a similar facility for doing input and output to debugging devices.

class Vdbostream

VxD developers often use a terminal (or terminal emulator) connected to a serial port for a debugging display in order to view the primary display while debugging. The class **Vdbostream** allows you to perform output by way of the serial port in much the same way you do output with iostreams. For example, consider the following fragment.

```
dout << "The VM handle is " << (DWORD)hVM << endl;
```

Soft-ICE users: You can use dout to send output to Soft-ICE/W; it will appear in the command window of the Soft-ICE/W display.

This has the expected result, namely, emitting by way of the serial port the quoted string, followed by the value of variable *hVM* formatted in hexadecimal, and terminated by a carriage return and newline sequence. The identifier *dout* is an instance of class **Vdbostream** the VTOOLSD class library provides.

The rules for output using operator << to instances of **Vdbostream** are as follows.

Data Type	Action
char	The data is emitted as a single ASCII character.
char*	The data is emitted as a null terminated string of ASCII characters.
unsigned long DWORD	The data is emitted as eight hexadecimal digits, padded with zeroes on the left.
unsigned short WORD	The data is emitted as four hexadecimal digits, padded with zeroes on the left.
unsigned char BYTE	The data is emitted as two hexadecimal digits, padded with zeroes on the left.
All others	Illegal for class Vdbostream. You can cast to one of the supported types or derive a class to meet your requirements.

class Vmonostream

Class **Vmonostream** supports stream output to the monochrome display. The formatting and syntax rules are inherited from **Vdbostream**.

Vmonostream has additional member functions for getting and setting the cursor position and for clearing the display. See **INCLUDE\VDEBUG.H** for specifications of these functions. The VTOOLSD class library provides a default instance of class **Vmonostream** named *dmono.*

You must add the string "DEBUGMONO=TRUE" to the [386enh] section of **SYSTEM.INI** in order for this class to work.

Here is an example of how you could use this class.

```
dmono.clearScreen();

dmono << "In event handler, ref data=" << dwRef << " last
error="

     << wLastErr << endl;
```

class Vdbistream

Class **Vdbistream** supports input from the debugging terminal attached to the serial port. The class supports the same data types that **Vdbostream** supports. On input, a carriage return or newline terminates a string. A character other than a hexadecimal digit terminates a numeric value. The VTOOLSD class library supplies a default instance of class **Vdbistream** named *din.*

Here is an example of how you could use this class.

```
dout << "Enter the value: ";

din >> dwValue;
```

Using Vdbostream as a Base Class

You will recall the goal of the example program we are developing is to output a message when a virtual machine is created or destroyed. We intend our message to go to DBWIN, however, and not simply to the debugging terminal or monochrome display.

Our example VxD will use a class *derived from* **Vdbostream** to send output to DBWIN. Just as **Vmonostream** overrides the protected virtual function **_output** to send output to the monochrome display, we can create a class that overrides **_output** to send output to DBWIN.

There are several benefits in taking this approach. First, it leverages the formatting capability and natural syntax already implemented in **Vdbostream**. Second, the result of our effort will be a new class, **Vdbwinstream**, which will serve as a reusable software component. The class will be usable in any VxD you develop, not only this example. Finally, the development of this class will illustrate the usage of some other classes in the VTOOLSD class library.

We can now make a start at defining our new class, **Vdbwinstream.**

```
class Vdbwinstream : public Vdbostream
```

```
{
protected:
   virtual VOID _output(const char* p);
};
```

Notice we override the virtual member function **_output**, whose argument is a pointer to a string to be output. Now the task at hand is to figure out how to implement this function so the string is sent to DBWIN.

We certainly do not want to force VxDs that use **Vdbwinstream** to wait until the message finds its way as far as DBWIN. Rather, we would like to simply write it to a buffer where it can be processed later by our helper application, RELAY.EXE. To fulfill this need, we will take advantage of another class in the VTOOLSD class library, namely, **VPipe**.

Pipe Classes

It is often necessary to transfer data from one virtual machine to another or to delay processing a stream of data. One way to do this is with a *pipe*. A pipe is an object that supports data transfer using read and write methods that operate on a "first in, first out" basis. In the Windows environment, where each virtual machine is in its own address space, a pipe is a convenient mechanism for constructing a data channel between applications or between a VxD and an application. The VTOOLSD class library provides two classes you can use to implement pipes in your VxD.

The first class, **VPipe**, is a base class that provides low level pipe functionality. This includes pipe I/O, locking for exclusive access, and notifications. All member functions are overridable so derived classes can tailor the pipe's behavior as required.

The second class, **VDosToWinPipe**, is designed to provide a data channel from a V86 mode DOS application to a 16-bit Windows application. **VDosToWinPipe** is derived from **VPipe**. You will find the discussion of this class later in this chapter. The tutorial does not make use of this class.

class VPipe

This class provides a byte stream data transfer mechanism that includes pipe I/O, locking for exclusive access, and notification of data availability. The pipe this class implements is essentially a "first in, first out" queue for an unstructured byte stream.

Member Functions

VPipe	Constructor
~VPipe	Destructor
lastError	Get error status
read	Read data from pipe
write	Write data to pipe
lock	Lock for exclusive access
unlock	Unlock exclusive access
OnDataAvailable	Notification function called when pipe becomes not empty
OnSpaceAvailable	Notification function called when pipe becomes not full

Using class VPipe

Although this is mainly a base class, understanding its structure is useful in working with derived classes.

The primary member functions are **read** and **write**, which move data out of and into the pipe. Before making any modifications to the pipe, both **read** and **write** invoke member function **lock** to ensure exclusive access.

Locking for Exclusive Access

In the base class, locking is implemented by a simple guard bit. Calls to the I/O member functions return **PIPE_BUSY** if the internal call to **lock**FUNCVPIPELOCK returns FALSE, i.e., if the pipe was locked at the time of the call. Classes derived from **VPipe** that override **lock** (and **unlock**) can choose to take some action other than simply returning FALSE when the pipe is already locked. For example, they can choose to block the caller on a semaphore object until the pipe is available (cf. **VDosToWinPipe**).

Member Functions for Event Notification

Class **VPipe** provides two member functions that implement notification of data available for pipe readers and notification of space in the pipe for pipe writers.

If the pipe is empty and member function **write** is called with a non-zero count, then, after the new data has been inserted into the pipe, member function **write** invokes overridable member function **OnDataAvailable.** Similarly, when a read operation causes the pipe to go from being completely full to not completely full, member function **read** calls overridable member function **OnSpaceAvailable**. These member functions are provided in order to allow derived classes to receive notification of significant events.

Consider a client who is reading data from the pipe. If member function **read** returns zero, the client does not want to tie up system resources by making repeated read attempts on the empty pipe. If the pipe class in question overrides **OnDataAvailable**, the client can arrange to be notified when there is data available. **OnDataAvailable** is not called every time something is written to the pipe; it is only called when something is written to an empty pipe.

Using VPipe as a Base Class

The design of **VPipe** is intended to allow extensibility and ease of use. The behavior of any pipe is largely determined by how the locking and notification mechanisms work. Derived classes can override any of the member functions in order to implement more sophisticated locking or notification schemes.

For our tutorial example, we will incorporate **VPipe** as an additional base class of **Vdbwinstream** and override **VPipe**'s member function **OnDataAvailable**. The pipe will serve as a buffer for messages sent to the stream. We need to override **OnDataAvailable** so we can respond when the pipe object sends notification that there is a message in the pipe.

At this stage, the declaration of our class looks like this.

```
class Vdbwinstream : public Vdbostream, public VPipe

{

protected:

   virtual VOID _output(const char* p);

   virtual VOID OnDataAvailable();

};
```

Now we can implement our member function **_output**; it will simply invoke member function **write** to insert the string into the pipe.

```
VOID Vdbwinstream::_output(const char* s)

{

        write((char*)s, strlen(s)+1);

}
```

The next task we face is deciding what to do when **OnDataAvailable** is called. We know its job ultimately will be to communicate with the helper application, RELAY.EXE. Since this is a Windows application running in the system VM, it can directly communicate with a VxD only when the system VM is active. If **Vdbwinstream** is to be a generally reusable class, we do not want to require its users to be running in the system VM.

What we need to do is arrange to have the VMM call us when the system VM is active. To do this, **Vdbwinstream**::**OnDataAvailable** will make use of another class from the VTOOLSD class library, namely, **VVMEvent**.

Before getting into the details of **VVMEvent**, we need to examine the general techniques for handling various kinds of system events. In the following discussion, keep in mind that the word "event" is used in two senses.

- First, there are events in the broader sense, encompassing interrupts, faults, pressing a hot key, and several others.
- Then there is a set of VMM services that refer to *events*, upon which classes such as **VVMEvent** are built. In this narrower sense, an event is a function callback the VMM makes at the request of a VxD. The type of the event determines the conditions under which the VMM makes this callback.

Subscribing to Events

Much of what a VxD does is process events. Depending on what purpose the driver serves, it might need to hook an interrupt, trap an I/O port, or catch all page faults. A large number of services the VMM provides are for allowing VxDs to be notified when such events occur.

An important function of the class library is to provide a standardized way of subscribing to events. Once you have hooked an interrupt, it should be easy to hook a fault, a hot key, access to an I/O port, or other similar events.

Here is the basic approach for subscribing to events using the VTOOLSD class library.

- Determine which base class from the library is appropriate for the event. The event processing classes are listed below.
- Derive a class from the base class. Your class must define a constructor and override member function **handler**.
- Create the constructor. There are generally class specific parameters you pass through to the base class constructor.
- Create a handler function to perform whatever actions are needed when the event occurs.
- Use operator **new** to generate an instance of the derived class.
- Call member function **hook** and test the Boolean result for success. (In some event classes, **hook** is replaced by another function or functions whose names are more suggestive of the operation in question.)

Here is an example for class **VHotKx**.

In an include file, declare your class as derived from the base class, with member functions being the constructor and the handler**:**

```
class CtrlCEvent : public VHotKey

{

public:
```

```
            CtrlCEvent();
            virtual VOID handler(BYTE, keyAction_t,
    DWORD,DWORD, DWORD);
    };
```

Place the member functions in a code module. The constructor simply calls the base class constructor and the handler can perform a variety of functions.

```
CtrlCEvent::CtrlCEvent() :
        VHotKey(SCAN_CODE_C, SCAN_NORMAL, HKSS_Ctrl, CallOnPress)
{
}
CtrlCEvent::handler(BYTE scan, keyAction_t ka, DWORD shift,
                DWORD refData, DWORD elapsed)
{
        // whatever your handler does
}
```

In the initialization code of your VxD, create an instance of the class with operator **new** and call member function **hook**.

```
CtrlCEvent* pCC = new CtrlCEvent();
if (pCC->hook())
// hot key is hooked
else
// failed to hook the hot key
```

Each of the classes listed below follows (or nearly follows) the above usage guidelines. With each class is listed the type of event with which the class corresponds.

VPreChainV86Int	Interrupt occurring in V86 mode
VInChainV86Int	Interrupt occurring in V86 mode
VInChainPMInt	Interrupt occurring in protected mode
VHardwareInt	Hardware interrupt
VHotKey	Hot Key
VProtModeFault	Fault occurring in protected mode application
VV86ModeFault	Fault occurring in V86 mode application
VVMMFault	Fault occurring in VMM or VxD

VPreChainV86Int	Interrupt occurring in V86 mode
VInvalidPageFault	Unhandled page fault exception
VDeviceAPI	Invocation of another VxD's PM or V86 API Entry
VGlobalEvent	Global Event
VVMEvent	Virtual Machine Event
VThreadEvent	Thread Event
VPriorityVMEvent	Priority VM Event
VV86Callback	Callback from V86 mode
VProtModeCallback	Callback from protected mode
VIOPort	Access to a particular I/O address
VGlobalTimeOut	Notification of elapsed time interval
VVMTimeOut	Notification of elapsed time interval of VM execution

Now that you have seen the general approach for handling various kinds of events using the VTOOLSD class library, we will examine a specific case relevant to the development of the tutorial.

Event Classes

There are certain junctures when the system is not in the appropriate state for your VxD to perform some operation. For example, you might be restricted from making VMM calls, you might not be in the correct virtual machine context, or you might want to defer to a virtual machine performing a time critical operation. At these times, *events* provide a means for you to arrange with the VMM to be called back when the system is in the desired state. The term *event* in this context means an asynchronous invocation of a handler function by the VMM under conditions requested by some VxD. In general, you use events to defer some action to a later point in time when the system is in the appropriate state for that action to take place.

One common case where use of events is required involves hardware interrupt processing. If the VMM is interrupted by a hardware interrupt, the interrupt handler cannot make non-async VMM calls. What it can do is arrange for the system to call back a second handler at a point in time when you are not restricted from making non-async calls.

Another example of event usage involves actions that must take place in the context of a specific virtual machine. Using events, you can request to be called back the next time the virtual machine is active.

Because it is necessary to create events at interrupt level, the base class for events, **VEvent**, overrides operator **new**. This implementation of **new** uses a special heap of fixed sized blocks. When a block is allocated, a guard bit in that block is set to one. This structure makes it possible to safely reenter this implementation of **new**, and thereby makes it safe to allocate instances of classes derived from **VEvent** at any time. These instances are automatically deleted after member function **handler** returns.

The VTOOLSD class library provides the following event classes.

VGlobalEvent	Event to run after interrupt processing complete
VVMEvent	Event to run in a specific virtual machine context
VPriorityVMEvent	Event to run in a specific virtual machine with special options
VThreadEvent	Event to run in a specific thread context
VEvent	Base class for other event classes

This section discusses class **VVMEvent** and how it relates to the tutorial. You will find discussions of the other event classes towards the end of this chapter.

class VVMEvent

A VM Event is an event whose execution requires the system to be running in the context of a specific virtual machine. Furthermore, a VM Event resembles a global event in that the VMM will not invoke the event handler until interrupt processing is complete.

Member Functions

VVMEvent	Constructor
schedule	Arrange to have handler called back
call	Schedule invocation of handler or call handler immediately
cancel	Cancel previously scheduled invocation of handler
handler	Function to be invoked, i.e., the event handler

Using class VVMEvent

You use a class derived from **VVMEvent** if you need to perform operations in the context of a virtual machine that cannot be the current virtual machine. Here are the steps.

- Derive a class from **VVMEvent**. Your class must define a constructor and override member function **handler**.
- Create a constructor for your class that invokes the base class constructor, passing to it the parameters that define your VM event. The parameters are the handle of the virtual machine in which the event should occur and the reference data value, which is optional.

- Write the event handler for your class. The arguments passed to the handler are the handle of the current virtual machine, a pointer to the virtual machine's client register structure, and the reference data value passed to the constructor.
- In the initialization code for your VxD, call **VEvent::initEvents**.
- Create an instance of your event class using **operator new**. Base class **VEvent** allows allocation at interrupt level.
- After you create an instance of your class, you can call either member function **call** or member function **schedule**, both of which will eventually result in the invocation of member function **handler**.

Example

Getting back to our tutorial example, let us define the event **Vdbwinstream::OnDataAvailable** will create upon notification that there is data in the pipe. The reason we need an event is we want the VMM to notify us when the system VM is active so we can communicate with our helper application (a Windows app). We will use **VVMEvent** as a base class for our new class, **DBWSEvent** (you will find this declaration in **DBWINSTR.H**).

```
class DBWSEvent : public VVMEvent

{

public:

   DBWSEvent(Vdbwinstream* pDbws);

   VOID handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs, PVOID refData);

};
```

Note we are following the general procedure for the event processing classes. We have derived a class from the appropriate base class, defined a constructor, and overridden member function **handler**.

Now we need to write the constructor for **DBWSEvent**. Its single parameter is a pointer to the **Vdbwinstream** object that creates it. When the constructor for **DBWSEvent** invokes its base class constructor (for **VVMEvent**), it passes this pointer as the reference data parameter. Recall that the system passes the constructor's reference data parameter to the event handler when it is invoked. This is a convenient way for the **DBWSEvent**'s *handler* to obtain a pointer to the **Vdbwinstream** object. Here is the constructor.

```
DBWSEvent::DBWSEvent(Vdbwinstream * pDbws) :
                            VVMEvent(Get_Sys_VM_Handle(), (PVOID)pDbws)
{
}
```

Notice the first argument to the base class constructor will always be the handle of the system VM, because we always want the event we are creating to run in the context of that virtual machine.

Now that we have a constructor for the required event, we can go back and write the body of **Vdbwinstream::OnDataAvailable**. (Recall that we are leveraging the capability of class **VPipe** to notify us when it is time to communicate with the helper application.)

```
VOID Vdbwinstream::OnDataAvailable()
]

DBWSEvent* pEvent = new DBWSEvent(this);  // create the event
    if (pEvent) // if OK
    pEvent->call(); // tell VMM to
               //  schedule/call it
}
```

You might notice class VVMEvent does not strictly follow the convention of having a member function called **hook**. Instead, it has two functions that serve an analogous purpose, namely, **call** and **schedule**.

We have not yet written **DBWSEvent::handler**, which will be a very important function. As we shall see, it will use nested execution services to call the Windows API function **PostMessage**. The arguments we pass to **PostMessage** will send a message to RELAY.EXE, instructing it to read data from the pipe.

At this point, we should review where we stand with our tutorial example.

- We have set up the basic structure of our VxD using the framework classes, **VDevice** and **VVirtualMachine**.
- In order to implement message output to DBWIN, we have chosen to create a new reusable class, **Vdbwinstream**, which is based on **Vdbostream**. We override **Vdbostream**'s virtual member function **_output**. This will allow *iostream* style output to DBWIN.
- **Vdbwinstream** also incorporates **VPipe** as a base class. By providing buffering for output, this allows VxDs that use **Vdbwinstream** to continue to execute, instead of waiting until DBWIN has displayed the message.
- **Vdbwinstream** overrides **VPipe**'s member function **OnDataAvailable**, which is called when data is written to the pipe. **OnDataAvailable** creates an instance of a new class, **DBWSEvent**. The handler function for this class is called when the system VM is active and receives as a parameter a pointer to the **Vdbwinstream** instance that created it.

The next task is to provide a way for the helper application, RELAY.EXE, to communicate with **Vdbwinstream**. You will recall from the discussion of the framework classes that the VMM provides a mechanism applications can use to call into VxDs. However, remember we are building a software component that should be usable by any VxD. If we structured **Vdbwinstream** such that it required the standard application interface mechanism, it would conflict with all VxDs that used the interface for other purposes. A better solution is to use a vendor specific entry point.

Vendor Entry Point Classes

Many VxDs provide services to application software through a defined programming interface. The standard way to implement this is to use the V86 entry point and protected mode entry point mechanisms provided by the VMM. It is for exactly this reason that class **VDevice** provides member functions **V86_API_Entry** and **PM_API_Entry**, which you can override in the class you derive from **VDevice**. There is, however, an alternative way for a VxD to provide programming services which takes advantage of a DPMI (DOS Protected Mode Interface) convention.

The DPMI specification includes a means for applications to query for the presence of vendor specific services in the operating environment. Complete the following steps to perform the querey.

1 Set the AX register to 0x168A.

2 Set DS:(E)SI to point to a null terminated string that identifies the vendor services desired.

3 Issue INT 0x2F.

If the vendor services specified by the string pointed to by DS:(E)SI are present, then, on return from the interrupt, register AL is zero and ES:(E)DI holds the address of a callback. The application can then access the vendor services by making a far call to the callback address.

For Windows 3.x, the advantage of using the DPMI vendor specific entry point over the normal API entry point mechanism is that the former allows you to identify your VxD with a string of your choosing. If you use the standard API mechanisms, you must use a VxD ID assigned by Microsoft. Since there are a limited number of IDs available, it might be safer and easier to use the vendor specific method. (Note: we use the terms “vendor specific method” and “DPMI method” interchangeably.)

For Windows 95, the standard convention for API entry points is preferable to the DPMI method because a device ID is no longer required. Simply set AX=0x1684, BX=0, and ES:DI to point to the 8-character, space-padded, uppercase, ascii name of the device. Issue the INT 0x2F, and the entry point address is returned in ES:DI if the device is found. See **EXAMPLES\C\POSTMSG\PMAPP.C** for an example.

The disadvantage of the vendor specific method is it requires hooking INT 0x2F, which could have a small impact on system performance.

How to Add a Vendor Specific Service Entry Point to Your VxD

The VTOOLSD class library includes two classes, **VPMDPMIEntry** and **VV86DPMIEntry**, that make it very simple to use the DPMI method for providing services to applications. By constructing an instance of a class derived from one of these base classes, you implicitly hook INT 0x2F and associate a handler with a callback address the INT 0x2F trap returns to applications that invoke it.

Here are the steps.

- Derive a class from **VPMDPMIEntry** to provide services to protected mode applications or **VV86DPMIEntry** to provide services to V86 mode applications. Your class must define a constructor and override member function **handler**. Note the handler is descended from base class **VCallback**.
- Create a constructor for your class that calls the base class constructor, passing to it the string that identifies your VxD. Applications use this string when they query for the presence of your VxD using INT 0x2F/AX=0x168A. The second parameter to the base class constructor is an optional reference value that will be passed to the entry point.
- Create your handler function. This is the routine that will gain control when an application makes a far call to the callback address returned by the INT 0x2F trap. Do not confuse this with the handler for INT 0x2F that performs the string comparison; the base class provides this functionality transparently. Note you must invoke member function **setReturn** from inside your handler to modify the client stack for correct return from the callback.

- Your handler implements services for applications. Parameters are typically passed in client registers.
- During initialization of your VxD, create an instance of your class using operator **new**. Test Boolean data member **m_bStatus** to determine if construction was successful. If successful, the value of **m_bStatus** is non-zero.

Notes

The authors of VTOOLSD do not recommend using class **VPMDPMIEntry** in dynamically loaded VxDs.

You must construct **VV86DPMIEntry** objects during initialization of your VxD. A single instance provides services for all virtual machines.

You must construct **VPMDPMIEntry** objects during virtual machine initialization. Each virtual machine requires a separate instance. For the system virtual machine, you can construct the object in **OnDeviceInit** or **OnInitComplete**.

Remember to call member function **setReturn** from inside your handler.

Example

Class **VPMDPMIEntry** provides everything we need to complete the class of the tutorial example, **Vdbwinstream**.

We will simply incorporate **VPMDPMIEntry** into **Vdbwinstream** as the third base class. We choose the protected mode version because we want to communicate with a Windows application. We override virtual member **handler**, and our class definition now looks like this:

```
class Vdbwinstream: public Vdbostream, public VPipe, public
VPMDPMIEntry

{

public:

   Vdbwinstream();

protected:

   virtual VOID _output(const char* p);

   virtual VOID OnDataAvailable();

   virtual BOOL handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs,

             PVOID refData);

}
```

We now have one member function for each base class: **_output** from **Vdbostream**, **OnDataAvailable** from **VPipe**, and **handler** from **VPMDPMIEntry**.

Once we construct an instance of our class, applications will be able to invoke member function **handler**. Applications obtain the callback address for this function by issuing INT 0x2F with AX=0x168A and DS:SI pointing to the string constant that identifies the interface. To define a string constant to serve as the unique identifier for the interface, we add the following line to **DBWINSTR.H**:

```
#define DWINSTR_ID_STRING "VxD->DBWIN Class Version 1.0"
```

Now we can write the constructor for **Vdbwinstream**, and complete the example VxD.

```
Vdbwinstream::Vdbwinstream() : Vdbostream(),VPipe(DBWS_PIPESIZE),
                                  VPMDPMIEntry(DWINSTR_ID_STRING, 0)
{
    m_bActive = FALSE;    // TRUE only when helper app is running
}
```

The conclusion of the tutorial is presented in the next section.

Interfacing to Applications

In this section, we illustrate by example some general techniques VxDs can use to interact with applications.

Let us begin by looking at the finished declaration of our new class, **Vdbwinstream**.

```
class  Vdbwinstream : public Vdbostream, public VPMDPMIEntry,
                public VPipe
{
public:
   Vdbwinstream();


protected:
   virtual BOOL handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs,
             PVOID refData);
   virtual VOID _output(const char* p);
   virtual VOID OnDataAvailable();
```

```
    BOOL m_bActive;
    WORD m_PostMessageSeg;
    WORD m_PostMessageOff;
    WORD m_PostMessageHwnd;

    friend DBWSEvent;
};
```

Notice we have added some data members to the class. They are:

m_bActive	Boolean that indicates if there is an instance of the helper application, RELAY.EXE, running.
m_PostMessageSeg	Aapplication level selector of Windows API function PostMessage
m_PostMessageOff	Offset component of the address of PostMessage
m_PostMessageHwnd	hWnd to pass to PostMessage

You will recall we want **DBWSEvent::handler** to invoke **PostMessage** with arguments that will send a message to RELAY.EXE. We grant access to **Vdbwinstream**'s protected data members by making **DBWSEvent** a **friend**.

How does the **Vdbwinstream** instance learn the address of **PostMessage**? When RELAY.EXE starts, it passes the information to **Vdbwinstream::handler**.

That function looks like this.

```
BOOL Vdbwinstream::handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs,
                PVOID refData)
{
        char* pBuf;
        DWORD cbCount;

        switch (_clientAX)
        {
        case DBWS_REGISTER:// invoked when RELAY initializes
                m_PostMessageHwnd = _clientBX;
                m_PostMessageSeg  = _clientCX;
```

```
                m_PostMessageOff  = _clientDX;
                m_bActive = TRUE;
                _clientAX = 0;
                break;

        case DBWS_READ:// invoked when RELAY receives
 // command DBWS_EVENT
                pBuf = (char*)Map_Flat(CLIENT_ES, CLIENT_DI);
                cbCount = _clientBX;
                _clientCX = read(pBuf, cbCount);
                _clientAX = 0;
                break;

        case DBWS_EXIT:// invoked when RELAY exits
                m_bActive = FALSE;
                break;
        }

        setReturn();
        return TRUE;
}
```

We have defined three function codes to which the handler responds. RELAY.EXE loads AX with the function code and calls to the address returned by INT 0x2F/AX=0x168A. The function codes have the following meanings.

DBWS_REGISTER	Tells Vdbwinstream RELAY.EXE is running and supplies information needed to call PostMessage. CX:DX is the address of PostMessage, and BX is the main window handle of RELAY.EXE.
DBWS_READ	Requests data from the pipe. ES:DI points to the buffer and BX is the buffer size. On receipt of this message, the handler calls VPipe's member function, read.
DBWS_EXIT	Tells Vdbwinstream RELAY.EXE is terminating.

The next piece of the puzzle is **DBWSEvent::handler**. The VMM calls this handler when the system VM is the current virtual machine.

```
VOID DBWSEvent::handler(VMHANDLE hVM,CLIENT_STRUCT* pRegs,PVOID
refData)
{
        CLIENT_STRUCT saveRegs;
        Vdbwinstream* pDbwin = (Vdbwinstream*)refData;

        Save_Client_State(&saveRegs);
        Begin_Nest_Exec();

        Simulate_Push(pDbwin->m_PostMessageHwnd);       // hwnd
        Simulate_Push(WM_COMMAND);                      // message
        Simulate_Push(DBWS_EVENT);                      // wParam
        Simulate_Push(0);                               // lParam
        Simulate_Push(0);
        Simulate_Far_Call(pDbwin->m_PostMessageSeg,
             pDbwin->m_PostMessageOff);
        Resume_Exec();
        End_Nest_Exec();
        Restore_Client_State(&saveRegs);
}
```

The event handler uses nested execution services to invoke **PostMessage**. The reference data parameter was set in **Vdbwinstream::OnDataAvailable** to the address of the **Vdbwinstream** instance that created the **DBWSEvent**. Notice we must call **Save_Client_State** and **Restore_Client_State** around the nested execution.

Class **Vdbwinstream** is now complete. If you look at its code in **DBWINSTR.CPP** and **DBWINSTR.H**, you will see there is not that much to it. It inherits most of its functionality from the base classes. You can use this code in other VxDs you are debugging, in conjunction with **RELAY.EXE** and **DBWIN.EXE**.

Here is the final form of CLASSTUT.CPP, the main module of our example.

```
// CLASSTUT.CPP - main module for VxD CLASSTUT
//
```

```
// Copyright (c) 1998, Compuware Corporation. All rights reserved

#define DEVICE_MAIN
#include "classtut.h"
#include "dbwinstr.h"

Declare_Virtual_Device(CLASSTUT)

Vdbwinstream* pDwin;

DWORD ClasstutDevice::CLASSTUT_GetVersion()
{
   return CLASSTUT_Major << 8 | CLASSTUT_Minor;
}

BOOL ClasstutDevice::OnDeviceInit(VMHANDLE hVM, PCHAR szCmdTail)
{
        VEvent::initEvents();
        pDwin = new Vdbwinstream;
        return TRUE;
}

BOOL ClasstutDevice::OnCreateVM(VMHANDLE hVM)
{
        *pDwin << "VM created, handle = " << (DWORD)hVM << endl;
        new ClasstutVM(hVM);
        return TRUE;
}

ClasstutVM::ClasstutVM(VMHANDLE hVM) : VVirtualMachine(hVM) {}
```

```
VOID ClasstutVM::OnDestroyVM()
{
        *pDwin << "VM destroyed, handle = " << (DWORD)m_handle
               << endl;
        delete this;
}
```

A few notes on this function.

- Notice if you use **Vdbwinstream**, you must first call **VEvent::initEvents**. This initializes the special heap from which instances of classes derived from **VEvent** are allocated.
- Notice we do not declare a global static instance of **Vdbwinstream**; instead we allocate it dynamically. In general, you should not statically allocate instances of classes whose constructors invoke VMM services.
- Finally, notice **ClasstutVM::OnDestroyVM** deallocates the instance of **ClasstutVM**. There is no need to keep the instance around after control message DESTROY_VM has been sent.
- When you build CLASSTUT.VXD, you must use the *debug version* of the class library. To test the tutorial example, do the following:
  - In the [386enh] section of **SYSTEM.INI**, add the line "device= <*path*>\CLASSTUT.VXD", where <*path*> stands for the directory in which **CLASSTUT.VXD** resides.
  - Restart the debug version of Windows.
  - Run **DBWIN.EXE**.
  - Run **RELAY.EXE**.

When you create a new DOS box or launch a DOS application from Windows, you will see a message appear in the DBWIN window.

Memory Management

Classes discussed in this section:

VpageObject	Base class for allocating objects on unlocked pages
VlockedPageObject	Base class for allocating objects on locked pages
VGlobalV86Area	Base class for allocating objects in first 1 MB of virtual machine's address space

VPageBlock	Resizable block of linearly contiguous committed pages
VV86Pages	Linearly contiguous range (in first 1 MB) of pages whose properties are modifiable and for which page faults can be hooked

Heap Classes

Whenever you invoke operator **new**, you are allocating memory. Similarly, when you invoke operator **delete**, you are freeing memory. The heap from which memory is allocated and freed depends on which operator **new** and operator **delete** functions are active in the scope of the object being allocated or freed.

The Default Heap

The VTOOLSD class library is structured such that, for the vast majority of objects, the implementations of operator **new** and operator **delete** access memory in the system heap provided by the VMM. VMM services **HeapAllocate** and **HeapFree** provide the basic services upon which the default operator **new** and operator **delete** are based. The default heap is locked and uninitialized.

The Page Heap

The default heap is appropriate for objects of size less than 4 KB. For larger objects, it is preferable to allocate memory in a page granular manner. VTOOLSD provides two base classes, **VPageObject**, and **VLockedPageObject**, that make this possible. Both these classes override operator **new** and operator **delete** with functions that use the VMM's page services to allocate and free memory. If you have a class with a data size greater than 4 KB, you should specify either **VPageObject** or **VLockedPageObject** as a base class. By doing so, you cause memory for the class to be allocated using page services, instead of the default system heap.

For Example:

```
// this class has a large data size
class BiDirPipe : public VPageObject
{
public:
    BiDirPipe();
    int read(char* buf, int count);
    int write(char* buf, int count);
protected:
    char inputBuffer[4000];
```

```
    char outputBuffer[4000];
    char* readptr;
    char* writeptr;
};

// allocate memory with page services
//    because VPageObject is base class
BiDirPipe* pPipe = new BiDirPipe();
```

The difference between **VPageObject** and **VLockedPageObject**, as the names suggest, is the objects derived from the latter are locked, i.e., they cannot be swapped to disk by the virtual memory manager. Since physical memory pages are a precious system resource, you should restrict your use of them. Use locked pages for objects that can be accessed during hardware interrupts.

The base class for these page-based objects increases the data size of derived classes by four bytes.

The V86 Global Area Heap

VTOOLSD provides one additional heap of a highly specialized nature. This heap allows you to allocate memory in the first linear megabyte of virtual machines. For classes derived from **VGlobalV86Area**, operator **new** invokes the VMM service **Allocate_Global_V86_Data_Area**. This reserves data storage addressable by code running in the virtual machine as well as from your VxD. These objects *can only be created during initialization* and cannot be freed.

The base class does not add any additional storage requirement to the derived class.

Use of this class *requires an extra parameter* to operator new. The parameter supplies numerous options, including the ability to maintain separate instances of the data in each virtual machine. See the on-line documentation for details.

The data size of classes derived from **VGlobalV86Area** should be a multiple of 4096. **VGlobalV86Area** itself has a data size of zero.

Member Functions

operator new	Allocates storage in V86 address space
operator delete	Does nothing -- there is no deallocation function

For Example:

```
class XBuffer : public VGlobalV86Area
{
char b[4096];
};

XBuffer* xb = new(GVDAPageAlign | GVDAReclaim) XBuffer;
```

Page Arrays

Sometimes what you need to do is explicitly allocate a set of pages, without overriding operator **new.** VTOOLSD provides two distinct classes for this purpose: **VPageBlock** and **VV86Pages**.

Class **VPageBlock** is a set of linearly contiguous committed pages. It provides member functions that allow you to resize, lock, and unlock pages. You can obtain the linear address of the block and the physical address of any page in the block. Use this class to create large resizable buffers or to supply a pool of committed pages for use with class **VV86Pages**.

Class **VV86Pages** is a contiguous range of pages in the first megabyte of virtual machine address space. Member functions allow you to specify properties (writable/read-only, present/not present, etc.) for pages in the range. Furthermore, you can hook all page faults in the range to the fault handler associated with the class. Thus, you can derive a class from **VV86Pages** that allows you to manage all accesses to a range of V86 address space. An example usage of this class is to manage the address space associated with a display device.

class VPageBlock

This class provides a block of linearly contiguous memory pages. Parameters to the constructor allow you to finely control the characteristics of the pages. Use this class to create large resizable buffers or to supply a pool of committed pages for use with class **VV86Pages**.

Member Functions

VPageBlock	Constructor
VPageBlock	Destructor
reallocate	Grows or shrinks the block
getLinearBase	Retrieves the linear base address of the block
getPhysicalAddress	Gets the physical address of a page in the block
getSize	Retrieves the size of the block, in pages
lock	Locks one or more pages of the block in memory
unlock	Unlocks one or more pages of the block
getAvailable	Determines the size of largest available block
discard	Discards contents of contiguous linear pages

For Example:

```
// allocate a block of 32 locked pages for virtual machine hVM

VPageBlock* pPagePoolObject =
new VPageBlock(32, PG_VM, PAGELOCKED, hVM, 0, 0, 0, 0);

if (  (pPagePoolObject != 0) &&
  (pPagePoolObject->getSize() != 0) )
{
// VPageBlock was successfully constructed

PCHAR pPool = (PCHAR)pPagePoolObject->getLinearBase();
memcpy(pPool, buffer, nBytes);

}
else
// failed to construct VPageBlock object
```

class VV86Pages

The purpose of this class it to allow you to manage all access to a range of V86 address space. For example, you could use this class to manage the address space for a display device.

Member Functions

VV86Pages	Constructor
assign	Reserve the range for exclusive use by your VxD
deassign	Release the range from exclusive use
hook	Hook all page faults in the range to member function handler
setProperties	Modify the properties of one or more pages in the range
mapPhys	Map a physical page into the range
map	Map a page from a VPageBlock into the range
unmap	Unmap pages by mapping in the system null page
handler	Handle page faults in the range

Using class VV86Pages

Class **VV86Pages** uses the VTOOLSD event processing conventions for trapping page faults in an address range. In addition, it supplies member functions that allow you to modify the logical and physical characteristics of memory in the managed range.

Here are the steps you need to take.

1. Derive a class from **VV86Pages**. Your class must define a constructor and override the member function **handler**.

2. Create a constructor for your class that invokes the base class constructor, passing to it parameters that define the range of memory you are managing.

3. Create a page fault handler for your class. The arguments the VMM passes to the handler are the handle of the virtual machine that caused the page fault and the absolute page number (a value between 0 and 0x10F, *not* an index relative to the base of the region).

4. Your page fault handler is responsible for correcting the fault. You can either change the properties of the faulting page with member function **setProperties** (e.g., change it from read-only to writable), or map a committed page into the range at the faulting address with member functions **map** or **mapPhys**. Alternatively, your handler can modify the state of the offending virtual machine, e.g., change its CS:(E)IP, such that when it resumes execution the fault will not occur. If the handler does not take some measures to correct the fault, the fault will reoccur and the handler will again be immediately invoked.

5 During initialization, create an instance of your class.

6 To claim the range, and reserve it for exclusive use by your VxD, call member function **assign**. If this fails, the memory range is already in use for some other purpose. This step is not strictly required, but is recommended as a means to avoid conflict with other VxDs.

7 Assignments can be global, i.e. they apply to all virtual machines, or local, i.e., they apply to a single virtual machine. You can make global assignments at any time, most commonly during initialization. However, you cannot make local assignments during initialization.

8 Hook all page faults in the range by calling member function **hook** and test the result for success.

9 Set the properties of the pages with member function **setProperties**. The properties you can set are paired: **PP_WRITABLE** or **PP_READONLY**, **PP_PRESENT** or **PP_NOTPRESENT**, and **PP_USER** or **PP_SYSTEM**.

The properties are defined as bit masks and can be combined. If neither property of a pair is specified, the corresponding property is not modified. Setting conflicting properties has no effect.

You must specify a virtual machine when you change a property. Accordingly, **VVirtualMachine::OnVMInit** is a convenient place to initialize the range of pages for a virtual machine.

To ensure your handler is invoked when the range is accessed, you can set the page properties to **PP_NOTPRESENT.** For example.

For Example:

The following code goes in the include file for your class.

```
class MonochromeTextSpace : public VV86Pages
{
    MonochromeTextSpace();
    virtual VOID handler(DWORD iPage, DWORD cPages);
};
```

The following code goes in the implementation file for your class.

```
MonochromeTextSpace::MonochromeTextSpace() :
            VV86Pages(0xB0, 2)
{
    // any additional initialization here
```

```
}
VOID MonochromeTextSpace::handler(VMHANDLE hVM,DWORD linPgNum)
{
    // whatever your handler does
}
```

The following code goes in the initialization code for your VxD.

```
MonochromeTextSpace* pMonoRegion=new MonochromeTextSpace();
if (pMonoRegion == 0)
    // allocation error

// assign in all virtual machines
if (pMonoRegion->assign(0))
    // assignment successful
else
    // unable to assign region to VxD

if (pMonoRegion->hook())
    // all page faults in range invoke member function
    // handler REGARDLESS OF WHICH VIRTUAL MACHINE CAUSES
      // THE FAULT
else
// failed to hook the pages
```

The following code goes in your virtual machine initialization code.

```
// set first two pages of range to present and read-only
pMonoRegion->setProperties(hVM, 0, 2, PP_PRESENT +
PP_READONLY);
```

Interrupt Classes

The VTOOLSD class library provides the following classes to support handling interrupts.

VSharedHardwareInt	Use this class if you are writing a VxD to support hardware that requires interrupt processing and the interrupt can be shared between devices. If the device is a PCI bus device, use this class.
VHardwareInt	Use this class if you are writing a VxD to support hardware that requires interrupt processing. The class provides an interface to the virtual programmable interrupt controller, VPICD, which allows you to virtualize the IRQ.
VPreChainV86Int	Use this class to process interrupts that occur in V86 mode. The handler is invoked prior to any application level handlers. These interrupts can only be hooked at initialization time.
VInChainV86Int	Use this class to process interrupts that occur in V86 mode. When you hook the interrupt, VTOOLSD modifies the interrupt vector table of the current virtual machine to point to a callback that will invoke your handler. The parent class, where much of the functionality resides, is VInChainInt.
VInChainPMInt	Use this class to process interrupts that occur in protected mode. When you hook the interrupt, VTOOLSD modifies the virtual interrupt descriptor table of the current virtual machine to point to a callback that will invoke your handler. The parent class, where much of the functionality resides, is VInChainInt.

The DAA class for hardware interrupts is **KInterrupt**.

class VHardwareInt and class VSharedHardwareInt

VTOOLSD provides the class **VHardwareInt** to facilitate hardware interrupt processing, and, specifically, to abstract the virtualization of IRQs, described below. A basic understanding of VPICD is useful as a prerequisite to successful use of the class **VHardwareInt**.

Member Functions

VHardwareIn	Constructor
VHardwareInt	Destructor
OnHardwareInt	Called when hardware interrupt occurs
OnVirtualInt	Called when VPICD invokes handler in virtual machine
OnVirtualEOI	Called when virtual machine issues EOI
OnVirtualIRET	Called when virtual machine returns from handler
OnVirtualMask	Called when virtual machine changes interrupt mask
hook	Virtualizes an IRQ
unhook	Unvirtualizes an IRQ
assertt	Asserts a virtual IRQ in a virtual machine
deassert	Deasserts a virtual IRQ in a virtual machine
getStatus	Retrieves partial state information for a virtualized IRQ
getCompleteStatus	Retrieves full state information for a virtualized IRQ
testPhysicalRequest	Tests the state of the physical IRQ signal
sendPhysicalEOI	Causes issuing of EOI to the physical PIC
physicalMask	Masks an IRQ on the physical PIC
physicalUnmask	Unmasks an IRQ on the physical PIC
setAutoMask	Instructs the PIC to perform intelligent IRQ masking
convertIntToIRQ	Converts an interrupt number to an IRQ number
convertIRQToInt	Converts an IRQ number to an interrupt number
forceDefaultOwner	Controls default processing for an IRQ

Background

In any multitasking environment, management of hardware interrupts is an important system level function. The system must dispatch hardware events to the correct drivers and applications and must allow specialized device drivers to provide customized interrupt service in a well-behaved fashion.

Under Windows, the Virtual Programmable Interrupt Controller Device (VPICD) is responsible for initial handling of all hardware interrupts. It provides a default mechanism that invokes interrupt handlers residing in virtual machines and it allows other VxDs to override the default interrupt processing mechanism as required.

In all PC bus architectures, there are a set of interrupt request signals (IRQs). When a hardware device asserts an IRQ while interrupts are enabled, an interrupt occurs and VPICD gains control of the processor. For a given IRQ, VPICD will either provide default handling or will allow another VxD to virtualize the IRQ.

In the default case, VPICD distinguishes between interrupts masked at Windows initialization time and those that are unmasked. Those that are unmasked are referred to as global, and VPICD invokes the handler in the currently active virtual machine when these interrupts occur. Initially, masked interrupts that subsequently become unmasked are "owned" by the virtual machine that unmasks them. VPICD invokes the handler in the owning virtual machine when these interrupts occur.

The default interrupt processing provided by VPICD consists of disabling interrupts and scheduling an invocation of the handler in the appropriate virtual machine. VPICD handles the return from the handler and restores the interrupt and trace flags to their previous states for the virtual machine in question.

IRQ Virtualization

When the default interrupt processing VPICD provides is not adequate or appropriate to support a piece of hardware, a VxD can "virtualize" an IRQ. By doing so, the VxD assumes responsibility for determining how the hardware interrupt is expressed to virtual machines. Specifically, a virtualizing VxD must:

- Determine which virtual machine's interrupt handler, if any, is to be called in response to the physical interrupt;
- Manage the virtual IRQ level for virtual machines;
- Control the rate of virtual interrupts into virtual machines in order to avoid stack overflow;
- Instruct VPICD to issue EOI to the physical PIC;
- Instruct VPICD to mask or unmask the physical interrupt as needed.

A VxD that virtualizes an IRQ receives notifications from VPICD during hardware interrupt processing. The events for which VPICD notifies the VxD are as follows:

- **Hardware interrupt -**VPICD notifies the virtualizing VxD the interrupt has occurred. This is the earliest notification of a hardware interrupt available under Windows. In addition to managing how virtual machines interact with the hardware interrupt, a virtualizing VxD can perform time-critical operations on the hardware at this time.
- **Virtual interrupt** - A virtualizing VxD requests VPICD to assert the virtual IRQ in a virtual machine, usually in response to a hardware interrupt notification. VPICD notifies the VxD when the virtual machine's handler is about to execute.
- **End of interrupt** - VPICD traps the issuing of EOI by the virtual machine and notifies the virtualizing VxD.
- **Interrupt return -** VPICD gains control when the handler returns and notifies the virtualizing VxD. This event is useful for pacing virtual interrupts to avoid stack overflows.
- **Mask change** - VPICD notifies the virtualizing VxD when a virtual machine changes the interrupt mask for the IRQ in question.

Handling Interrupts at Ring 0

Many VxDs that handle hardware interrupts do so primarily in order to minimize the time interval between the assertion of the IRQ and the time when the device is serviced. Such VxDs can elect not to express the interrupt in any virtual machine at ring 3.

If you are taking this approach, you only need to handle notification of the hardware interrupt and you can safely ignore the other notifications that VPICD provides. When VPICD invokes your interrupt handler, the handler can perform whatever operations to service the device and then ask VPICD to issue End-Of-Interrupt. A common strategy is to manage data buffers in the interrupt handler and use a global event to signal other code when the buffers become empty or full.

Using class VHardwareInt

VHardwareInt follows the VTOOLSD conventions for event processing. Here are the steps:

1 Derive a class from **VHardwareInt**. Your class must declare a constructor and you must override each of the member functions corresponding to notifications you wish to receive. Overriding of member function **OnHardwareInt** is mandatory. The other notification functions are:

`OnVirtualInt`	Called when VPICD invokes handler in virtual machine
`OnVirtualEOI`	Called when virtual machine issues EOI
`OnVirtualIRET`	Called when virtual machine returns from handler
`OnVirtualMask`	Called when virtual machine changes interrupt mask

2 Write your constructor. Usually, all that is required is invocation of the base class constructor with arguments appropriate to the IRQ you are virtualizing.

3 Write your notification handlers. Be careful to identically match the function declaration of the base class. Failure to do so will define a new member function instead of overriding the notification handler and you will not receive the notification.

4 For IRQs masked when Windows initialized, you might want the default state to be unmasked in new virtual machines as they come into existence. To effect this, call static member function **forceDefaultOwner** with the *hVM* parameter set to zero. This call must precede the call to member function **hook**.

5 Create an instance of your hardware interrupt class using operator **new**. This is almost always done in **VDevice::OnDeviceInit**, which you should override in your device class.

6 Call the member function **hook** and test the return value for success. After it is hooked, you might want to call **physicalUnmask** to ensure that the interrupt is unmasked.

Notes

You use other member functions of class **VHardwareInt** inside your notification handlers to control invocation of interrupt handlers in virtual machines. To raise the interrupt for a particular virtual machine, use member function **assert**. Use **deassert** to terminate or cancel an interrupt request for a virtual machine. You can instruct VPICD to control the physical PIC by using member functions **testPhysicalRequest**, **sendPhysicalEOI**, **physicalMask**, and **physicalUnmask**.

The VTOOLSD class library resolves the virtual function addresses of the notification functions when you call **hook** in order to minimize overhead and avoid a virtual function dispatch at hardware interrupt time.

Example

The following code goes in the include file for your class.

```
class XMyIRQ : public VHardwareInt
{
public:
XMyIRQ();
virtual VOID OnHardwareInt(VMHANDLE);
virtual VOID OnVirtualInt(VMHANDLE);
virtual VOID OnVirtualEOI(VMHANDLE);
virtual VOID OnVirtualIRET(VMHANDLE);
};
```

The following code goes in the implementation file for your class.

```
XMyIRQ::XMyIRQ() : VHardwareInt(MyIRQNumber, 0,0,0) {}


VOID XMyIRQ::OnHardwareInt(VMHANDLE hVM)
{
        assert(hVM);   // cause VM's int handler to run
        ClearCarry();  // tell VPICD interrupt
     // was handled
}
VOID XMyIRQ::OnVirtualEOI(VMHANDLE hVM)
{
```

```
        deassert(hVM);// drop virtual IRQ
// level for VM
        sendPhysicalEOI();
}
```

The following code goes in the initialization code for your VxD.

```
XMyIRQ* pIRQ = new XMyIRQ();
if (pIRQ->hook())
{
// IRQ is virtualized
}
else
{
// failed to virtualize IRQ
}
```

class VPreChainV86Int

Use this class to process interrupts that occur in V86 mode. The handler is invoked prior to any application level handlers.

Member Functions

VPreChainV86Int	Constructor
~VPreChainV86Int	Destructor
hook	Hooks the interrupt to the handler
unhook	Unhooks the interrupt from the handler
enablePostChainHandler	Configures invocation of post-chain handler
handler	Pre-chain interrupt handler
postChainHandler	Post-chain handler

Using class VPreChainV86Int

This class is used primarily for software interrupts. For hardware interrupts that require specialized processing at the IRQ level, use class **VHardwareInt**. The class might be used for hardware interrupts if the default hardware interrupt processing of VPICD provides correct and sufficiently fast service.

This class follows the conventions of the VTOOLSD class library. Here are the steps.

1 Derive a class from **VPreChainV86Int**. Your class must define a constructor and override the member function **handler**. If you wish to trap the interrupt both before and after application level handlers, then you must also override member function **postChainHandler**.

2 Write a constructor that invokes the base class constructor, passing it the number of the interrupt you are hooking.

3 Write your handler functions.

4 Create an instance of your class using operator **new**.

5 Call the member function **hook** and test the result for success.

Example

The following code goes in the include file for your class.

```
class VDosInt : public VPreChainV86Int

{

public:

        VDosInt();

       virtual BOOL handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs,
DWORD dw);

        virtual VOID postChainHandler (
          VMHANDLE hVM,
          CLIENT_STRUCT* pRegs,
          PVOID refData);

};
```

The following code goes in the implementaion file for your class.

```
VDosInt::VDosInt() : VPreChainV86Int(0x21) {}


BOOL VDosInt::handler(VMHANDLE hVM,CLIENT_STRUCT* pRegs, DWORD
dw)

{

        switch (_clientAH)

        {

        case 0x3f:
```

```
        enablePostChainHandler();

        return FALSE;


VOID VDosInt::postChainHandler(VMHANDLE hVM, CLIENT_STRUCT*
pRegs, PVOID ref)
```

The following code goes in the inilialization code for your VxD.

```
VDosInt* pDosEvent = new VDosInt();

if (pDosEvent && pDosEvent->hook())

   // interrupt was successfully hooked

else

   // failed to hook interrupt
```

Post-chain Handlers

The system calls member function **handler** prior to calling any handlers installed at application level. It is often useful to examine the client state after all application level handlers have executed. For example, if you are monitoring some software interrupt, you might need to know when it is invoked (determined in your pre-chain handler) and what the results were (determined in your post-chain handler).

If **handler** returns TRUE, the VMM does not call any additional interrupt handlers. If **handler** returns FALSE, the VMM continues calling handlers, including those at application level.

Class **VPreChainV86Int** includes features that facilitate this kind of interrupt processing. To utilize it, override member function **postChainHandler** in your class definition. Then, in your member function **handler**, make a call to member function **enablePostChainHandler** and then return FALSE. This will cause the system to invoke your post-chain handler after all application level handlers execute. At this point, you can examine registers and memory of the virtual machine.

class VInChainInt

Class **VInChainInt** is a base class for **VInChainV86Int** and **VInChainPMInt**. These classes allow you to install an interrupt handler invoked via a callback from application level.

Member Functions

VInChainInt	Constructor
~VInChainInt	Destructor
hook	Hooks the interrupt to the handler
unhook	Unhooks the interrupt from the handler
setReturn	Modifies client stack to simulate IRET instruction
chainToPrevious	Modifies client stack to chain to previous handler

class VInChainV86Int

Class **VInChainV86Int** is derived from both **VInChainInt** and **VV86Callback**, which together comprise most of the functionality of the class.

You use this class to process interrupts that occur in V86 mode. The handler is invoked via a callback whose address is inserted into the V86 interrupt table. In other words, the VxD level handler is effectively chained in the application handler chain and can be invoked before and/or after application level interrupt handlers have been invoked. If the V86 interrupt table vector for the corresponding interrupt is written after the member function **hook** is called, the handler will not be called unless chained to by the handler that was installed after it.

Class **VPreChainV86Int** is generally preferable to this class because its handler is called regardless of how the interrupt vector table is modified and it requires fewer privilege level transitions. In addition, class **VPreChainV86Int** provides a method for trapping the return from the application level handlers. Use class **VInChainV86Int** if you wish to emulate the behavior of a DOS application or TSR that hooks into the interrupt chain. Unlike **VPreChainV86Int**, you can call the member function **hook** at any time, not only during initialization.

Member Function

VInChainV86Int	Constructor

Using class VInChainV86Int

This class follows the VTOOLSD conventions for event processing. Here are the steps.

1. Derive a class from **VInChainV86Int**. Your class must define a constructor and override the member function **handler**.

2. Write your constructor. It is usually sufficient to simply invoke the base class constructor, passing it the interrupt number.

3 Write your handler. The handler for this class is defined in the base class **VCallback**. Remember you must call either **setReturn** or **chainToPrevious** in order to set the client virtual machine's stack and registers to the desired state.

4 Create an instance of your class using operator **new**.

5 Call member function **hook**, and check the return value for success.

Example

The following code goes in the include file for your class.

```
class XDosInt : public VInChainV86Int
{
public:
   XDosInt();
   virtual BOOL handler(VMHANDLE hVM,
            CLIENT_STRUCT* pRegs, PVOID intNumber);
};
```

The following code in the implementation file for your class.

```
XDosInt::XDosInt() : VInChainV86Int(0x21) {}

BOOL XDosInt::handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs,
          PVOID intNumber)
{
        dout << "INT 21 ax=" << pRegs->CWRS.Client_AX
             << " bx=" << pRegs->CWRS.Client_BX
             << " cx=" << pRegs->CWRS.Client_CX
             << " dx=" << pRegs->CWRS.Client_DX
             << endl;

        chainToPrevious();
        return TRUE;
}
```

The following code goes in the inilialization code for virtual machines.

```
XDosInt* pInt = new XDosInt();
if (pInt->hook())
    // interrupt is hooked
else
    // failed to hook interrupt
```

class VInChainPMInt

Class **VInChainPMInt** is derived from both **VInChainInt** and **VProtModeCallback**, which together comprise most of the functionality of the class.

You use this class to process interrupts that occur in protected mode. The handler is invoked via a callback whose address is inserted into the virtual interrupt descriptor table of a virtual machine. In other words, the VxD level handler is effectively chained in the application handler chain and can be invoked before and/or after application level interrupt handlers have been invoked. If the interrupt is hooked by an application after the member function **hook** has been called, the handler will not be called unless chained to by the handler that was installed after it.

Member Function

VInChainPMIn	Constructor

Using class VInChainPMInt

This class follows the VTOOLSD conventions for event processing. Here are the steps.

1. Derive a class from **VInChainPMInt**. Your class must define a constructor and override the member function **handler**.
2. Write your constructor. It is usually sufficient to simply invoke the base class constructor, passing it the interrupt number.
3. Write your handler. The handler for this class is defined in the base class **VCallback**. Remember you must call either **setReturn** or **chainToPrevious** in order to set the client virtual machine's stack and registers to the desired state.
4. Create an instance of your class using operator **new**.
5. Call member function **hook**, and check the return value for success.

Example

Refer to the example for **VInChainV86Int** above.

DMA Classes

Background

Direct Memory Access, commonly referred to as DMA, is a means to transfer data without the assistance of the processor. The DMA Controller, a hardware component of the standard PC architecture, drives the address and data buses to effect the transfer and can interleave bus cycles with the processor. High speed devices such as hard disks or frame grabbers are most likely to utilize DMA.

In the standard architecture, there are two DMA Controllers, which together provide seven channels. Each channel can be individually programmed for a transfer operation independent of other channels. Drivers for devices that utilize DMA must avoid attempts to utilize the same channel. Assignment of devices to channels is generally by convention, although in Plug and Play environments software arbitration can be used.

To perform DMA, drivers first issue I/O instructions that program the physical addresses and direction of the transfer and then "unmask" the Controller to start the transfer. A Controller can be programmed to issue an interrupt when the transfer completes.

Virtualization by VDMAD

Under Windows, multiple virtual machines must share the DMA hardware in the system. The Virtual DMA Device, VDMAD, *virtualizes* the DMA Controllers to make this possible. By trapping all access to the DMA Controllers, VDMAD allows code running in a VM to operate as if it had exclusive access to one or more DMA channels.

For each virtual machine, VDMAD maintains a *virtual state* of each DMA channel. The virtual state is the information required to place the channel in the physical state corresponding to the sequence of programming operations executed in the VM.

When a VM issues the instruction to unmask the Controller, and thereby start a transfer, VDMAD must ensure that the memory is physically contiguous because the Controller deals exclusively with physical addresses. To perform the actual transfer, VDMAD might have to copy the memory to or from a physically contiguous buffer it maintains internally. These operations are transparent to the ring 3 driver.

Virtualization by Other VxDs

For some devices, the virtualization of the DMA Controller VDMAD provides might not be adequate. To accommodate this, VDMAD allows other VxDs to virtualize the Controller on a per channel basis.

When a VxD virtualizes a DMA channel, VDMAD notifies that VxD whenever ring 3 I/O operations modify the virtual state of that channel. The VxD must supply a handler to receive the notification.

This handler can utilize services provided by VDMAD to do any of the following:

- Query the virtual state of the channel.
- Modify the virtual state of the channel.
- Modify the state of the physical controller.
- Invoke VDMAD's default handler.

The available options offer flexibility. A handler can do as little as merely query the virtual state and pass the event down to VDMAD for default processing or can choose to assume full responsibility for the event, including direct manipulation of the physical Controller.

Buffers and Regions

VxDs that implement full virtualization of a DMA channel must use additional services provided by VDMAD for managing *regions* and *buffers*. A region is a block of linearly contiguous memory that either contains data being transferred to the hardware device or receives data transferred from the hardware device. If the region is not physically contiguous, or does not meet certain other criteria, then the VxD must obtain a *buffer* from VDMAD in order to perform the transfer.

VDMAD provides services for associating a region with a DMA channel, locking a region, and transferring data to and from buffers.

Classes for DMA Support

The VTOOLSD class library provides two classes for DMA support: **VDMAChannel**, which is a virtualized DMA channel, and **VDMABuffer**, a DMA buffer.

class VDMAChannel

Class **VDMAChannel** is used as a base class by VxDs that support a hardware device for which the default virtualization of DMA channels provided by VDMAD is not appropriate. The virtual member function **handler** is called by VDMAD whenever a ring 3 driver modifies the virtual state of the channel. Additional member functions control behavior of the virtual channel and provide access to the physical DMA Controller.

Member functions

VDMAChannel	Constructor
~VDMAChannel	Destructor
getVirtualState	Fetch the transfer parameters for the channel
setVirtualState	Sets the transfer parameters for the channel
setPhysicalState	Sets the mode on the physical controller
disableTranslation	Disables address translation when ring 3 code uses Virtual DMA Services
enableTranslation	Instructs VDMAD to convert passed addresses per VM memory map
getEISAMode	Fetches EISA mode parameter
setEISAMode	Sets EISA mode parameter
physicalMask	Inhibits physical channel
physicalUnmask	Allows physical transfer on channel
getRegion	Fetch previously configured region parameters
setRegion	Specifies region parameters
hook	Hooks channel modification events to member function handler
unhook	Unhooks channel modification events from member function handler
handler	Handles all modifications to virtual state of channel made by ring 3 code

Using class VDMAChannel

This class follows the event processing conventions of the VTOOLSD class library. All VxDs that choose to virtualize a DMA channel should derive a class from **VDMAChannel**. The class should define a constructor and override member function **handler**. You create an instance of the class derived from **VDMAChannel** during initialization, using operator **new**. The initialization code must then call member function **hook** and test the Boolean return value for success. If successful, all accesses to the DMA controller at I/O addresses that correspond to the channel specified by the constructor's parameter will result in a call to member function **handler**.

The usage of **VDMAChannel**'s member function **handler** is very dependent upon the particular needs of the hardware in question. The underlying services provided by VDMAD allow a great deal of flexibility in how a DMA channel is virtualized.

Some VxDs can assume total responsibility for the DMA channel. In this case, the member functions that directly modify the physical channel are most relevant, namely **setRegion**, **setPhysicalState**, **physicalMask**, and **physicalUnmask**.

Other VxDs might require only limited modifications to the virtualization provided by VDMAD. In this case, the handler typically calls **getVirtualState** and **setVirtualState** and then passes the event on to VDMAD by invoking **VDMAChannel::handler**.

Still other VxDs might hook the DMA channel, but use hardware other than the DMA controller to effect the transfers programmed by the ring 3 code.

Example

The following code appears in the include file.

```
class MyDMAChannel : public VDMAChannel
{
public:
    MyDMAChannel();
    VOID handler(VMHANDLE hVM):
};
```

The following code appears in the implementation of the class.

```
// virtualize channel 7
    MyDMAChannel::MyDMAChannel() : VDMAChannel(7) {}

VOID MyDMAChannel::handler(VMHANDLE hVM)
{
    // . . . whatever your handler does
}
```

The following code appears in the initialization code for the VxDs.

```
MyDMAChannel* pMyDMA = new MyDMAChannel();
if (pMyDMA->hook())
  // DMA channel is virtualized, handler will be called
  // when ring 3 code sends I/O to controller for channel
else
  // failed to virtualized channel - not available
```

Here is the skeleton of a handler that invokes the base class handler.

```
VOID MyDMAChannel::handler(VMHANDLE hVM)
{
```

```
    // examine and/or modify virtual state of channel
    // . . .

    VDMAChannel::handler(hVM); // let VDMAD do all the work
}
```

class VDMABuffer

VDMAD provides buffer services to support DMA transfers when the DMA region is not physically contiguous or cannot be locked. Class **VDMABuffer** abstracts these services.

Member functions

VDMABuffer	Constructor
~VDMABuffer	Destructor
copyFrom	Copies data from buffer to region
copyTo	Copies data to buffer from region
reserve	Reserves buffer space (static member)

Using class VDMABuffer

If your VxD will require a DMA buffer to support a device, you should call static member function **reserve** during initialization in order to inform VDMAD of your device's buffer requirements. Call **reserve** only inside **OnSysCriticalInit**.

To obtain a buffer, create an instance of class **VDMABuffer**. You must not create the **VDMABuffer** until immediately before it is to be used. You should destroy the object as soon as the DMA operation is complete, because VDMAD buffers are a limited resource and might be required by other devices. If the lifetime of a **VDMABuffer** object is within a single function, as is usually the case, the object can be allocated on the stack. Use of operator **new** is *not* necessary. The object's destructor is called automatically at the closing brace (}).

When you transfer data from memory to a hardware device, use member function **copyTo** prior to enabling the DMA operation. This function copies data from the DMA region to the buffer, from where it can then be successfully transferred to the device.

When you transfer data from a hardware device to memory, use member function **copyFrom** after the DMA operation completes. This function copies the data transferred from the device into the buffer to the DMA region.

Example

This fragment illustrates a DMA operation that transfers data from a device to memory:

```
void TransferFromDevice()
{
// attempt to lock region
bStatus = VDMAChannel::lockRegion( . . . );
// if not lockable, use a buffer
if (!bStatus)
{
   VDMABuffer myBuffer(region, size);
// construct the object
   if (myBuffer.m_createError == 0)  // zero means OK
   {
// use VDMAChannel members to
// program the transfer
       myChannel.setRegion(&myBuffer, FALSE,
region, size, myBuffer.m_physAddr);
// . . . enable the transfer
       myChannel.physicalUnmask();
// . . .
// after operation completes, copy from buffer to region
       myBuffer.copyFrom(region, size,
0, errorCode);
   }
} // closing brace destroys VDMABuffer object
else
{
// use locked region
}
}
```

Callback Classes

Callbacks act like gateways between applications and VxDs. Each callback associates an address in application space with a function in VxD space. When an application jumps to or calls the address in application space, the VxD function gains control.

class VCallback

The class **VCallback** is an internal base class for **VV86Callback** and **VProtModeCallback**. Its definition is found in **VCALLBAK.H**. A **VCallback** object provides a means for an application to call a function in a VxD.

The callback classes differ from other classes with **handler** functions in that there is no member function called **hook**. Callbacks are installed by simply creating an instance of the class.

Note there is no destructor because the VMM does not provide a means to deallocate callbacks. Therefore, they should be considered a valuable resource. Allocate the callbacks your VxD needs once, during initialization, as opposed to allocating one each time it is needed.

Member Functions

handler	VxD level function invoked when application calls to the callback address.
getAddr	Returns the application level address that applications call to invoke the handler function.
setReturn	Modifies the client stack so application level call returns correctly.

class VV86Callback

The class **VV86Callback** provides a means for a V86 mode application to directly invoke a function in a VxD.

Member Functions

VV86Callback	Constructor

Usage

Use a class derived from **VV86Callback** to cause a V86 mode application to invoke a function in your VxD. Each **VV86Callback** you create is associated with a unique segment:offset in the V86 address space. Calling or jumping to that address causes control to pass to member function **handler** of the class you derive from **VV86Callback**. The syntax of the handler is defined by the base class **VCallback**. Create an instance of your class and obtain the callback vector with the member function **getAddr**. VxDs typically insert the callback vector at strategic locations in the V86 address space as a means to trap application-level operations.

You typically do not use a VV86Callback to provide a means to explicitly call your VxD. Instead, you should use the V86 mode API capability. See class **VDevice**.

Note **VV86Callback** is a base class for **class VInChainV86Int**. Use this class instead of inserting the callback vector into the V86 interrupt vector table of a virtual machine.

Example

```
class XMSTrap : public VV86Callback
{
public:
XMSTrap();
virtual BOOL handler(VHHANDLE hVM,
      CLIENT_STRUCT* pRegs,
      DWORD refData);
DWORD m_savedXMSvector;
};
```

class VProtModeCallback

The class **VProtModeCallback** provides a means for a protected mode application to directly invoke a function in a VxD.

Member Functions

VProtModeCallback	Constructor

Using class VProtModeCallback

Use a class derived from **VProtModeCallback** to cause a protected mode application to invoke a function in your VxD. Each **VProtModeCallback** you create is associated with a unique selector:offset in the protected mode address space. Calling or jumping to that address causes control to pass to the handler function of the class you derive from **VProtModeCallback**. The syntax of the handler is defined by the base class **VCallback**. Create an instance of your class and obtain the callback vector with the member function **getAddr**. VxDs typically insert the callback vector at strategic locations in the application's address space as a means to trap application level operations.

You typically do not use a VProtModeCallback to provide a means to explicitly call your VxD. Instead, you should use the protected mode API capability. See class **VDevice**.

Note **VProtModeCallback** is a base class for class **VProtModeInt**. Use this class instead of inserting the interrupt vector to the callback vector.

Fault Classes

The VTOOLSD class library provides classes that facilitate hooking faults. You can process these different kinds of faults:

VV86ModeFault	Faults occurring in V86 mode
`VProtModeFault`	Faults occurring in protected mode
`VVMMFault`	Faults occurring at the VxD level (ring 0)
`VNMIEvent`	Non-maskable interrupt event
`VInvalidPageFault`	Page faults not processed by the system
`VV86Pages`	Page faults in a specified range of V86 address space

By deriving a class from one of the above base classes, you implicitly associate a handler function with methods for hooking and unhooking the fault.

The base class of the first four fault classes above is **VFault**. This class declares the member functions for hooking and unhooking faults, as well the member function that is the fault handler.

VInvalidPageFault does not have **VFault** as its base class, because its handler is called differently and does not require a constructor. Use that class if your VxD needs to process page faults.

VV86Pages is also distinguished from the other fault classes. It is used to manage all memory access to particular ranges of the V86 address space. The class includes other member functions for managing pages. An example usage of this class is to manage the address space associated with a display device, such as a VGA. Use this class if your VxD needs to control access to a specific portion of the low linear megabyte in all virtual machines.

Using the Fault Classes

The fault classes follow VTOOLSD conventions for event processing. Here are the steps for hooking a fault:

- Depending on the kind of fault you wish to hook, derive a class from either **VProtModeFault**, **VV86ModeFault**, **VVMMFault**, or **VNMIEvent**. Your class must define a constructor and override the member function **handler**.
- Define a constructor for your class that calls the base class constructor, passing the fault number (fault number not required for classes derived from **VNMIEvent**).
- Override member function **handler** with the handler for your fault.
- At the appropriate point (usually during initialization), create an instance of your fault class with operator **new**.
- Call the member function **hook**. If it returns TRUE, the system will invoke the handler when the fault occurs.
- When you wish to unhook the fault, call member function **unhook**.

Example

The following code goes in the include file for your class.

```
class My_V86_GPFault : public VV86ModeFault
{
public:
    My_V86_GPFault();// constructor
    virtual BOOL handler(VMHANDLE, CLIENT_STRUCT*, int);
};
```

The following code goes in the implementation file for your class (0xD is the fault number for General Protection).

```
My_V86_GPFault::My_V86_GPFault() : VV86ModeFault(0xD) {}
```

```
BOOL My_V86_GPFault::handler(VMHANDLE h,CLIENT_STRUCT*
pRegs,int i)

{

// whatever your handler does

}
```

The following code goes in the initialization code for your VxD.

```
// create instance

My_V86_GPFault* pGP = new My_V86_GPFault();

// hook the fault

bStatus = pGP->hook();
```

class VFault

Member Functions

VFault	Constructor
~VFault	Destructor
hook	Hooks a fault event to a handler
unhook	Unhooks a fault event from a handler
handler	Handles the fault event

Using class VFault

VFault is used only as a base class for other fault classes.

class VV86ModeFault

Member Functions

VV86ModeFault	Constructor

Using class VV86ModeFault

This class follows the general event processing scheme used by many classes in the class library.

Class **VV86ModeFault** is a base class used if your VxD processes faults that occur in V86 mode. Examples of faults are Invalid Opcode Fault, General Protection Fault, and Divide by Zero Fault.

The class is used only as a base class. You derive a class from **VV86ModeFault** that declares a constructor and overrides the member function **handler**. You create an instance of the derived class using the **new** operator and call the member function **hook**.

Call the member function **unhook** when you no longer want to process faults. The destructor for **VV86ModeFault** unhooks the handler before destroying the object.

class VProtModeFault

Member Functions

VProtModeFault	Constructor

Using class VProtModeFault

This class follows the general event processing scheme used by many classes in the class library.

Class **VProtModeFault** is a base class you use if your VxD processes faults that occur in protected mode applications, including Windows applications. Examples of faults are Invalid Opcode Fault, General Protection Fault, and Divide by Zero Fault.

The class is used only as a base class. You derive a class from **VProtModeFault** that declares a constructor and overrides the member function **handler**. You use the **new** operator to create an instance of the derived class and call the member function **hook**. This is typically done during initialization.

Call the member function **unhook** when you no longer want to process faults. The destructor for **VProtModeFault** unhooks the handler before destroying the object.

class VVMMFault

Member Functions

VVMMFault	Constructor

Using class VVMMFault

This class follows the general event processing scheme used by many classes in the class library.

Class **VVMMFault** is a base class you use if your VxD processes faults that occur in VxDs. Examples of faults are Invalid Opcode Fault, General Protection Fault, and Divide by Zero Fault.

The class is used only as a base class. You derive a class from **VVMMFault** that declares a constructor and overrides the member function **handler**. You use the **new** operator to create an instance of the derived class and call the member function **hook**. This is typically done during initialization.

Call the member function **unhook** when you no longer want to process faults. The destructor for **VVMMFault** unhooks the handler before destroying the object.

class VNMIEvent

Member Functions

VNMIEvent	Constructor

Using class VNMIEvent

Class **VNMIEvent** is a base class you use if your VxD processes non-maskable interrupt events.

This class follows the general event processing scheme used by many classes in the VTOOLSD class library.

The class is used only as a base class. You derive a class from **VNMIEvent** that declares a constructor and overrides the member function **handler**. You create an instance of the derived class using the **new** operator and call the member function **hook**. This is typically done during initialization.

Call the member function **unhook** when you no longer want to process faults. The destructor for VVMMFault unhooks the handler before destroying the object.

Example

The following code goes in the include file for your class.

```
class NMI : public VNMIEvent
{
public:
        NMI();
        virtual BOOL handler(VMHANDLE hVM,
    CLIENT_STRUCT* pRegs,
              int iFault);
};
```

The following code goes in the implementation file for your class.`NMI::NMI() : VNMIEvent() {}`

```
BOOL NMI::handler(VMHANDLE hVM,
 CLIENT_STRUCT* pRegs,
 int iFault)
{
        // whatever your handler does
}
```

The following code goes in the initialization code for your VxD.

```
NMI* pNMI = new NMI;
if (pNMI && pNMI->hook())
    // successfully hooked
else
    // failed to hook
```

TimeOut Classes

A time-out object is a means for a VxD to arrange with the system to be notified after a specified time interval has elapsed. The TimeOut classes differ in how the associated time interval is measured by the system:

VGlobalTimeOut	Interval measured by system clock
VVMTimeOut	Time interval elapses only during execution of specified VM
VThreadTimeOut	Time interval elapses only during execution of specified Thread
VAsyncTimeOut	Like a global time out, but handler is called as soon as time expires, not on event cycle

Class **VTimeOut** is an abstract base class for the other TimeOut classes.

Member Functions

Set	Start timer countdown
Cancel	Cancel the time-out
handler	Function called when time expires

Using the TimeOut classes

The TimeOut classes follow the VTOOLSD conventions for event processing objects. They are used only as base classes. Your derived class defines a constructor and overrides member function **handler**. After constructing an instance of the object, you call member function **Set** to initiate the countdown of the timer. Each of the base TimeOut classes implements member function **Set** to request the desired type of time-out from the system.

Before calling **Set**, your VxD can assign any value to data member **m_refData** (type **PVOID**). This is a convenient way to carry data along with the object. Alternatively, your derived class can include additional data members.

The constructors for the different TimeOut classes have different calling sequences. Refer to the on-line documentation for details.

Calling member function **Set** only schedules the invocation of the handler after the time interval elapses; it does *not* suspend the calling VM.

After the time interval specified by a parameter to the constructor has elapsed, the system invokes member function **handler**. The arguments to the handler are the handles of the current virtual machine and current thread, a pointer to that virtual machine's register structure, and a value in units of milliseconds that indicates the length of the time interval between the expiration of the time-out and the invocation of the handler. The latter quantity arises from the granularity of the system clock, which is typically at least 20 milliseconds.

The system invokes member function **handler** during normal event processing. The handler is therefore *not* subject to the constraints that apply to hardware interrupts.

Example

The example below shows the implementation of a typical TimeOut class. It is based on **VGlobalTimeOut**, so the time interval is measured against the system clock. In addition to the time interval parameter, the constructor takes a pointer to a **VSemaphore** object as a parameter. The constructor stores the pointer in data member **m_refData**\ and the handler signals the semaphore when the time-out expires.

```
// class declaration - define constructor and override handler

class XTimer : public VGlobalTimeOut
```

```
{
public:
        XTimer(DWORD msec, VSemaphore* pSem);
        virtual VOID handler(VMHANDLE hVM, PVOID reserved,
                PCLIENT_STRUCT pRegs, DWORD dwLagTime);
};

// constructor - store the semaphore pointer as
// reference data member
XTimer::XTimer(DWORD msec, VSemaphore* pSem) :
VGlobalTimeOut(msec)
{
        m_refData = pSem;
}

// time-out handler - signal the semaphore when called
VOID XTimer::handler(VMHANDLE hVM, PVOID reserved,
                PCLIENT_STRUCT pRegs, DWORD dwLagTime)
{
        ((VSemaphore*) m_refData)->signal();
}
```

class VGlobalTimeOut

Class **VGlobalTimeOut** provides a time-out whose time interval elapses with the system-wide clock, i.e., it is not tied to a particular thread or virtual machine. Follow the VTOOLSD conventions for event processing classes; use this class as a base class, define a constructor that invokes the base class constructor, and override member function **handler**.

class VVMTimeOut

Class **VVMTimeOut** provides a time-out whose time interval elapses with the execution of a specified virtual machine, i.e., the timer expires when the accumulated execution time (after the call to member function Set) of the specified virtual machine reaches the time interval specified in the call to the constructor. Follow the VTOOLSD conventions for event processing classes; use this class as a base class, define a constructor that invokes the base class constructor, and override member function **handler**.

class VThreadTimeOut

Class **VThreadTimeOut** provides a time-out whose time interval elapses with the execution of a specified thread, i.e., the timer expires when the accumulated execution time (after the call to member function Set) of the specified thread reaches the time interval specified in the call to the constructor.

class VAsyncTimeOut

Note the handler for **VAsyncTimeOut** is called asynchronously, i.e., not at event time. This means the time-out is more accurate, but the handler cannot call most VMM/VxD services. The handler is subject to the same restrictions as a hardware interrupt handler.

VCOMM Classes

You can use the VTOOLSD class library to build a VCOMM port driver. Two cooperating classes, **VCommPortDriver** and **VCommPort**, simplify the creation of port drivers by managing the protocols that interface to VCOMM. This allows you to focus on the aspects of the driver that are specific to your hardware, rather than on the details of the VCOMM port driver architecture.

The Port Driver Class

Class **VCommPortDriver** is a base class you use to form a framework for a VCOMM port driver. Consistent with the fact that a port driver is a specialized type of virtual device, **VCommPortDriver** is derived from **VDevice**. Normally, VxDs built with the class library utilize **VDevice** as the base class for constructing the VxD framework. For port drivers, **VCommPortDriver** assumes this role.

The Comm Port Class

Class **VCommPort** abstracts communication ports. Its member functions correspond to the services that port drivers provide to VCOMM. For each type of port your port driver supports, you derive a class from **VCommPort** to implement an interface to that type of port. You override the various member functions of **VCommPort** to define the specific interactions between VCOMM and your hardware. The member functions you do not override in a derived class are assumed by VCOMM to be unimplemented services.

Operation

During initialization of the port driver, usually in **OnDeviceInit** or **OnSysDynamicDeviceInit**, the port driver creates instances of a class (or classes) derived from **VCommPort**. There should be one such instance for each physical port that the driver supports. After creating these port objects, the initialization code invokes **VCommPortDriver::Register**, which communicates the port information to VCOMM.

When applications (or other VxDs) request communication resources via VCOMM's exported services, VCOMM invokes member functions of the appropriate port objects in order to access the hardware. The class library automatically sets up all the linkage that makes this possible.

In summary, here are the tasks required to build a port driver:

1. Derive a class from **VCommPortDriver** (instead of its parent class, **VDevice**) to serve as your VxD framework class. You will need to override **OnDeviceInit** or **OnSysDynamicDeviceInit** in order to perform your initialization operations.

2. Derive a class from **VCommPort** and override the member functions that implement the interface to your hardware. VCOMM assumes services that correspond to member functions you do not override are unimplemented. The base class provides default functionality for **Setup**, but all others are the responsibility of the derived class.

3. In your initialization code, create instances of the class (or classes) you derived from **VCommPort**. There should be one such instance for each communication resource (physical port) your driver supports.

4. After creating the port objects, call **Register**. All subsequent interaction with VCOMM will occur via invocations of the member functions of the port objects you created.

class VCommPortDriver

Class **VCommPortDriver** is a specialized VxD framework class for VCOMM port drivers. It is designed to be used in conjunction with class **VCommPort**.

Member Functions

Register	Register driver with VCOMM
DoAcquireRelease	Instructs driver to use enable/release protocol for ports

Using class VCommPortDriver

This class plays exactly the same role as its parent class, **VDevice**, plays in other VxDs. As a framework class, one of its main functions (inherited from **VDevice**) is to provide linkage to control message handlers. For Windows for Workgroups 3.11, initialization generally occurs in **OnDeviceInit**. For Windows 95, port drivers need to override **OnSysDynamicDeviceInit** in order to process control message SYS_DYNAMIC_DEVICE_INIT.

In addition to providing control message dispatching, this class works with **VCommPort** to facilitate creation of VCOMM port drivers. When an object based on **VCommPort** is constructed, the constructor calls a private member of **VCommPortDriver.** This allows the driver to track all the port objects in its domain. During initialization, the port driver constructs port objects, and then calls member function **Register**.

Implementation

Member function **Register** calls VCOMM_Register_Port_Driver, passing the address of a private member function (DriverControl), which VCOMM then calls to instruct the driver to initialize. The initialization sequence differs slightly between Windows for Workgroups 3.11 and Windows 95. Under WfW 3.11, DriverControl is called exactly once. Under Windows 95, it is called separately for each physical port the Configuration Manager associates with the port driver. The association between the port driver and the port is defined via information in the **.INF** file when the port driver is installed. This information is recorded in the system registry.

For each port object, DriverControl calls VCOMM_Add_Port. A parameter to VCOMM_Add_Port points to a data structure inside the port object. This data structure contains a pointer to an array of procedure addresses. These procedures are the services VCOMM calls to request access to the port. DriverControl has set up this array to point to thunks that link to the overridden member functions of the class derived from **VCommPort**.

Example

Here is an include file for a VxD based on **VCommPortDriver**.

```
// SERIAL.H

#include <vtoolscp.h>


#define DEVICE_CLASS    SerialPortDriver

#define SERIAL_DeviceID  UNDEFINED_DEVICE_ID
```

```
#define SERIAL_Init_Order UNDEFINED_INIT_ORDER
#define SERIAL_Major      1
#define SERIAL_Minor      0

class SerialPortDriver : public VCommPortDriver
{
public:
   virtual BOOL OnSysDynamicDeviceInit();
};
```

Here is the main code module for the VxD.

```
#define DEVICE_MAIN
#include "serial.h"
Declare_Virtual_Device(SERIAL)
#undef DEVICE_MAIN

#include "comx.h"// defines VCommPort based class

BOOL SerialPortDriver::OnSysDynamicDeviceInit()
{
        new Comx("COM1");
        new Comx("COM2");

        Register();
        return TRUE;
}
```

class VCommPort

Class **VCommPort** provides an abstraction for a communication port resource. This class is used only in conjunction with **VCommPortDriver**.

Member Functions

VCommPort	Constructor
Initialize	Initialize the port
Open	Open the port
Close	Close the port
GetCommState	Get port communication status
GetProperties	Get port properties
SetCommState	Set port communication status
Setup	Configure port parameters
GetQueueStatus	Get status of port's transmit/receive queues
Purge	Purge data from queues
Read	Read data from port (or queue)
TransmitChar	Single byte write
Write	Write data to port
EnableNotification	Set VCOMM to notify driver of port events
EscapeFunction	
GetModemStatus	Get state of port's modem status bits
SetModemStatus	Set state of port's modem status bits
GetEventMask	Get port's current event mask
SetEventMask	Set port's event mask
SetReadCallback	Accept address of function to call on read event
SetWriteCallback	Accept address of function to call on write event
SetCommConfig	
GetCommConfig	
GetError	

Data Members

`m_data PortData`	Structure for this port (this is a structure, not a pointer)
`m_dcb DCB`	For this port (this is a *structure*, not a pointer)
`m_handle`	VCOMM port handle
`m_PortNumber`	Port number (passed to _VCOMM_Acquire_Port). Member function Initialize is responsible for setting this value (1-0x7f for COM ports, 0x80-0xff for LPT ports)

m_ownerVM	Handle of VM that currently owns port
m_name	Port name (pointer to null terminated ASCII string)

Using class VCommPort

For each *type* of port that your port driver supports, you need to derive a class from **VCommPort**. If all ports that your driver supports have the same behavior, a single class should suffice.

With exceptions noted below, the virtual member functions of **VCommPort** correspond to the standard set of services port drivers provide to VCOMM. By overriding member functions of **VCommPort**, you implicitly define the interaction between the VCOMM and the physical hardware. (In a non-object-oriented implementation, VCOMM passes a port handle to the driver to specify which port is the object of the request. The VTOOLSD class library provides linkage to invoke the appropriate member function of the port object in question).

While most member functions are invoked by VCOMM (via a two instruction thunk), member functions **Initialize**, **Open**, and **Close** are invoked by member functions of class **VCommPortDriver**.

To implement a useful driver, the class you derive from **VCommPort** must override most of the virtual functions. If you do not override a given virtual function, then VCOMM assumes the service corresponding to that function is unavailable. The exception to this rule is virtual member function **Setup**, which is implemented by the base class. This function, which sets up the transmit and receive queues, is not hardware specific. You can examine the source code to determine if what it does is appropriate for your driver, and, if it is not, override it.

Instances of your port class (or classes) should be created by the initialization code for your VxD, typically in **OnSysDynamicDeviceInit**. After creating the instances, call **Register** to register the port driver with VCOMM.

Example

Here is the class definition for a port object and a sample member function.

```
class Comx : public VCommPort
{
public:
   Comx(PCHAR name);

protected:
   virtual COMMREQUEST Initialize(DEVNODE handle, DWORD baseIO,
```

```
            DWORD irq);
   virtual COMMREQUEST Open(VMHANDLE hVM, PDWORD pError);


   virtual COMMREQUEST Close();
   virtual COMMREQUEST SetCommState(PDCB pDCB, DWORD
ActionMask);
   virtual COMMREQUEST GetCommState(PDCB pDCB);
   virtual COMMREQUEST GetProperties(_COMMPROP* pCommProp);
   virtual COMMREQUEST Setup(PCHAR RxQueue, DWORD cbRxQueue,
             PCHAR TxQueue, DWORD cbTxQueue);
   virtual COMMREQUEST ClearError(_COMSTAT* pComstat,
    PDWORD pError);
   virtual COMMREQUEST GetQueueStatus(_COMSTAT* pComstat);
   virtual COMMREQUEST Purge(DWORD qType);
   virtual COMMREQUEST Read(PCHAR buf, DWORD cbRequest,
      PDWORD pRxCount);
   virtual COMMREQUEST TransmitChar(CHAR ch);
   virtual COMMREQUEST Write(PCHAR buf, DWORD cbRequest,
PDWORD pTxCount);
   virtual COMMREQUEST EnableNotification(
PCOMMNOTIFYPROC pCallback,
          PVOID refData);
   virtual COMMREQUEST EscapeFunction(DWORD lFunc,
 DWORD InData,
           PVOID pOutData);
   virtual COMMREQUEST GetModemStatus(PDWORD pModemStatus);
   virtual COMMREQUEST GetEventMask(DWORD mask, PDWORD pEvents);
   virtual COMMREQUEST SetEventMask(DWORD mask, PDWORD pEvents);
   virtual COMMREQUEST SetModemStatusShadow(DWORD mask,
PBYTE pMSRShadow);
   virtual COMMREQUEST SetReadCallback(DWORD RxTrigger,
```

```
            PCOMMNOTIFYPROC pCallback,
DWORD refData);
   virtual COMMREQUEST SetWriteCallback(DWORD TxTrigger,
            PCOMMNOTIFYPROC pCallback,
DWORD refData);
};

COMMREQUEST Comx::Initialize(DEVNODE handle, DWORD baseIO,
DWORD irq)
{
#ifdef DEBUG
        dout << "Comx: initializing " << m_name << " Base I/O="
<< baseIO << " IRQ#=" << irq << endl;
#endif
        m_PortNumber = m_name[3]-'0';
        return TRUE;
}
```

Other Classes

class VIOPort

You use class **VIOPort** to virtualize a port. The VMM traps all application level I/O operations that are performed on virtualized ports and invokes a VxD level handler responsible for executing the operation.

If you just want to read or write data to or from a port in the I/O space, you do not need class **VIOPort**. Just use library functions _inp and _outp.

By deriving a class from **VIOPort**, you implicitly associate a handler function with methods for enabling and disabling trapping of I/O activity. Additional member functions of **VIOPort** allow you to control which virtual machines have trapping enabled for a given port.

Member Functions

VIOPort	Constructor
~VIOPort	Destructor
hook	Hook port activity to handler
unhook	Unhook port activity from handler
handler	Function invoked when I/O activity occurs
localEnable	Enable trapping of port in specified virtual machine
localDisable	Disable trapping of port in specified virtual machine
globalEnable	Enable trapping in all virtual machines
globalDisable	Disable trapping in all virtual machines

Using class VIOPort

Class **VIOPort** follows VTOOLSD conventions for event processing. Here are the steps:

- Derive a class from **VIOPort**. Your class must define a constructor and override the member function **handler**.
- Create a constructor for the class that invokes the base class constructor with a parameter to specify the port you want to trap.
- Create a handler for your class. The handler is responsible for virtualizing I/O operations on the port.
- Create an instance of your class using operator **new**. This is done during initialization.
- Call member function **hook** and test the result for success.

Example

The following code goes in the include file for your class.

```
class MyPort: public VIOPort
{
public:
   MyPort();
   virtual DWORD handler(VMHANDLE, DWORD port,
      CLIENT_STRUCT* pRegs, DWORD iotype,
   DWORD outdata);
};
```

```
#define MY_PORT_NUMBER 0x538
```

The following code goes in the implementation file for your class.

```
MyPort::MyPort() : public VIOPort(MY_PORT_NUMBER)
{
// initialize other class members, if any
}


DWORD MyPort::handler(VMHANDLE hVM, DWORD port,
       CLIENT_STRUCT* pRegs,DWORD iotype,
DWORD outdata)
{
    switch (iotype) {
        case BYTE_INPUT:
        .
        .
        .
    }
}
```

The following code goes in the initialization code for your VxD.

```
MyPort* pPort = new MyPort();
if (pPort->hook())
    // port activity is hooked to handler
else
    // failed to trap port
```

Notes

Create an instance of your class and call **hook** during initialization of your VxD. By default, the port will be hooked for all virtual machines. If this is not the desired behavior, call member function **globalDisable**. This will disable trapping in all existing virtual machines and set the default for new virtual machines to not trap the port. You reverse the effect of this call with **globalEnable**. You can also selectively enable and disable trapping of the port in particular virtual machines using **localEnable** and **localDisable**.

The member function **handler** that you supply must virtualize the port in question. This means I/O operations must be carried out or simulated by the handler transparently to the virtual machine that performs the operation. If your port supports string operations, your handler must affect changes to client registers holding the repeat count and offset values.

class VDeviceAPI

Many VxDs provide services to applications via the V86 API entry point and Protected Mode API entry point mechanisms (see class **VDevice** for more information). Occasionally, the need arises to intercept calls from an application to an API entry point of a known device. Doing so allows your VxD to monitor calls to another device's API entry point(s) and enhance or modify the functionality.

An instance of class **VDeviceAPI** corresponds to one or both the API entry points for a particular device. You use this class as a base class if your VxD needs to intercept calls from applications to some other VxD.

Member Functions

VDeviceAPI	Constructor
~VDeviceAPI	Destructor
hook	Hook member function handler to another device's V86 and/or PM API entry point
unhook	Unhook handler from V86 and/or PM API entry point
callPreviousHandler	Invoke original V86 or PM API entry point
handler	Function to handle V86 and/or PM API call

Using class VDeviceAPI

Class **VDeviceAPI** follows VTOOLSD conventions for processing events. Here are the steps.

1. Derive a class from **VDeviceAPI**. Your class must declare a constructor and override member function **handler**.
2. Create a constructor for you class that calls the base class constructor with parameters that specify the device APIs you wish to intercept.
3. Write the handler for you class. The system passes the handle of the current virtual machine and a pointer to the virtual machine's register structure.
4. Create an instance of your class using operator **new**. This is usually done during initialization of your VxD.
5. Call member function **hook** and test the result for success.

6 If your handler needs to invoke the original API entry point, use member function **callPreviousHandler**.

Example

The following code goes in the include file for your class.

```
// Protect mode API entry for virtual display device

class XVDDPMAPI : public VDeviceAPI

{

public:

   XVDDPMAPI();

   virtual VOID handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs);

};
```

The following code goes in the implementation file for your class.

```
XVDDPMAPI::XVDDPMAPI() : VDeviceAPI(VDD_DEVICE_ID,
DEVICE_PM_API) {}


VOID XVDDPMAPI::handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs)

{

        dout << "Grabber called VDD: Client AX = "

                << pRegs->CWRS.Client_AX

                << endl;


        callPreviousHandler(hVM, pRegs);

}
```

The following code goes in the initialization code of your VxD.

```
XVDDPMAPI* pVDDapi = new XVDDPMAPI();

if (pVDDapi && pVDDapi->hook())

        // successfully hooked

else

        // failed to hook
```

Notes

If you hook the API during initialization, make sure the device whose API you are intercepting has a lower initialization order than that of your VxD.

class VHotKey

You use the class **VHotKey** to trap specific keyboard events, typically the pressing of a key. By deriving a class from **VHotKey**, you implicitly associate a handler function with methods for defining, hooking, and unhooking your hot key.

Member Functions

VHotKey	Constructor
~VHotKey	Destructor
hook	Hooks the hot key to the handler
unhook	Unhooks the hot key from the handler
handler	Function invoked when hot key event is detected
reflectToVM	Pass hot key event through to a specified virtual machine
localEnable	Enable or disable hot key for a specified virtual machine
cancelState	Cancel hot key state (static member function)

Using class VHotKey

This class follows the VTOOLSD conventions for event processing. Here are the steps:

1. Derive a class from **VHotKey**. Your class must define a constructor and override the member function **handler**.
2. Create a constructor for your class that invokes the base class constructor to define your hot key.
3. Create a function to handle the hot key by overriding member function **handler** of **VHotKey**.
4. Create an instance of your class using operator **new**.
5. Call member function **hook** and test the return for success.

Example

The following code goes in the include file for your class.

```
class CtrlCEvent : public VHotKey
```

```
{
public:
        CtrlCEvent();
        virtual VOID handler(BYTE, keyAction_t,
          DWORD,PVOID, DWORD);
};
```

The following code goes in the implementation file for your class.

```
CtrlCEvent::CtrlCEvent() :
        VHotKey(SCAN_CODE_C, SCAN_NORMAL, HKSS_Ctrl,
CallOnPress)
{
}

CtrlCEvent::handler(BYTE scan, keyAction_t ka,
DWORD shift, PVOID refData,
DWORD elapsed)
{
        // whatever your handler does
}
```

The following code goes in the initialization code for your VxD.

```
CtrlCEvent* pCC = new CtrlCEvent();
if (pCC->hook())
    // hot key is hooked
else
    // failed to hook the hot key
```

class VList

The VTOOLSD class library contains a list class that encapsulates the VMM's Linked List Services. These services provide a convenient means for building linked lists and queues.

To build a list, you first create a **VList** object using the constructor. Then, for each item to be placed in the list, you allocate memory using member function **newNode** and attach the node to the list with member function **attach** or **attachTail**. Member functions **remove** and **removeFirst** remove items from the list. The memory for list items is deallocated with member function **deleteNode**.

Member Functions

VList	Constructor
~VList	Destructor
newNode	Allocates a new node
deleteNode	Deletes a node
attach	Attaches a node at the head of the list
attachTail	Attaches a node at the tail of the list
insert	Inserts a node after a specified node
first	Returns a pointer to the head of the list
next	Returns a pointer to the node following a given node
remove	Removes a node from the list
removeFirst	Removes the head of the list and returns a pointer to the removed node.

Using class VList

Items stored in lists are referred to as *nodes*. Memory allocation for nodes can only be performed by the list member function **newNode**. This allows the VMM to add the necessary overhead fields for linkage and to manage memory more efficiently. Likewise, deallocation of storage for nodes must be done with the list member function **deleteNode**. A **VList** can only store nodes of a single size, and, therefore, most likely of a single type.

If your list is to contain nodes that are simple scalars (e.g. a list of DWORDs or pointers), then you can simply call **newNode** and **deleteNode** to allocate and free nodes as necessary.

If, on the other hand, your list contains more complex objects, you can have a class that defines the type of object your list manages. This is referred to as the *node class*. If the type of object your list manages is a class, there are two important things you must add to that class in order to ensure correct allocation and deallocation of list nodes:

- Because storage for nodes *must* be allocated by the **VList** member function **newNode** and deallocated by the member function **deleteNode**, you should override the **new** and **delete** operators of the node class to call the node allocator and deallocators of the associated list.

- You should declare a static data member of type **VList*** that points to a list whose member functions can allocate and deallocate nodes.

For Example:

```
class MyNode //  define the node class
{
public:
   MyNode();
   void* _cdecl operator new(size_t n)   //override "new"
      {return m_pList->newNode();};
   void  _cdecl operator delete(void* p);//override "delete"
      {m_pList->deleteNode(p);};

   int myNodeDataA;
   int myNodeDataB;  // whatever you store in a node
   int myNodeDataC;

   static VList* m_pList; // pointer to a list for node
allocation
};

VList* VMyNode::m_pList; // declare outside of function { }
```

You must create a **VList** object for a node class before you create an instance of a node. When you create the list, store a pointer to it in the static member of your node class that points to the associated list, in this example, **m_pList**. This allows the **new** operator for the node class to call the **newNode** member function of its list.

For Example:

```
// FIRST, create a VList object and assign to list
// pointer of node class

VMyNode::m_pList = new VList(sizeof(VMyNode));

// THEN, create a node using the new operator
```

```
VMyNode* pNode = new VMyNode();
```

To delete a node, simply use the **delete** operator.

```
delete pNode;  // this will invoke VMyNode::m_pList-
>deleteNode().
```

For more examples of how to use **VList**, see the **VDOSKEY** example, which uses lists extensively.

Manipulating Lists

To add nodes to a list, use one of the following member functions:

attach	Attach a node to the head of the list
attachTail	Attach a node to the tail of the list (Note: attach is slightly faster than attachTail)
insert	Insert a node after a specified node

Neither of the above functions deallocates storage for the node; you must call deleteNode.

To remove nodes from a list, use one of the following member functions:

remove	Remove a specific node
removeFirst	Remove the head of the list (and return its address)

To iterate over a list, use the member function **first** to get the list head, then use the member function **next** to get successively a pointer to each list node.

class VSemaphore

The class **VSemaphore** is a simple encapsulation of the VMM semaphore services. Semaphores are generally used to manage resources shared by multiple virtual machines or threads.

Member Functions

VSemaphore	Constructor
~VSemaphore	Destructor
`wait`	Acquire controlled resource
`signal`	Release controlled resource
`signal_noSwitch`	Release controlled resource without reschedule
`signal_noSwitch`	Release resource but do not immediately reschedule threads

Using class VSemaphore

When you create a **VSemaphore** object, you specify a token count. The token count corresponds to the number of clients that can simultaneously acquire the resource in question.

Clients claim acquisition of the resource by calling member function **wait**. If the number of clients that currently hold the resource is less than the initial token count, then **wait** returns immediately and the caller holds the resource. Otherwise, the client is blocked by the VMM until another client releases the resource.

Clients release the semaphore by calling member function **signal** or the member function **signal_noSwitch (not available in version 3.1)**. The member function **signal** causes immediate rescheduling of virtual machines and threads, whereas **signal_noSwitch** does not.

Depending on your needs, you can create a **VSemaphore** object with a token count of zero and signal it as the corresponding resources become available. For example, a VM can wait for a message by blocking on a semaphore that was created with a zero token count A second VM can then signal the semaphore when it delivers a message.

In most cases it is necessary to allocate **VSemaphore** objects on the heap using the operator **new**. However, in cases where the semaphore is being used only within a single function, it is permissible to allocate it on the stack. In this case, before waiting on the semaphore you must somehow pass its address to the VM you intend to have signal the waiting VM.

The destructor for **VSemaphore**, invoked either by the operator **delete** (for objects created with **new**) or by a closing brace (for objects allocated on the stack), causes the semaphore to be destroyed. Any virtual machines that were blocked on the destroyed semaphore become schedulable.

Example

Here is a sketch of a set of routines that manage an array of packets shared by multiple virtual machines or threads.

```
VSemaphore* pSemHeap;
PACKET packets[PACKET_HEAP_SIZE];

void initPacketHeap()
{
   // create semaphore with token count corresponding to
   // resource size
   pSemHeap = new VSemaphore(PACKET_HEAP_SIZE);
   ...
}

PACKET* allocPacket()
{
   pSemHeap->wait(); // block if no packet available
   ...
   return pnext;
}

void freePacket(PACKET* p)
{
   pnext = p;
   ...
   pSemHeap->signal();  // signal that packet is available
}

void closePacketHeap()
{
   delete pSemHeap;  // release the semaphore
   ...
}
```

If your VxD uses more than one semaphore simultaneously, you must be careful to avoid a deadlock condition. This occurs when a thread blocks on a resource owned by the first thread. To avoid deadlock, enforce a hierarchical protocol of semaphore acquisition, i.e., require that the semaphores in your VxD be acquired in a specified order, and released in the reverse order.

class VMutex

The **VMutex** class encapsulates the VMM functionality for mutex objects. You use a VMutex object to guard a region of code against simultaneous execution by more than one thread.

Member Functions

VMutex	Constructor
~VMutex	Destructor
enter	Enter mutex region
leave	Exit mutex region
owner	Get owner of mutex
getHandle	Get handle of mutex
enterGlobal	Enter global critical section
leaveGlobal	Exit global critical section

Using class VMutex

Upon creation, a **VMutex** object is unowned. Calling the member function **enter** causes the calling thread to become the owner. Invoking **enter** of a mutex owned by another thread causes the calling thread to block. If the *owner* of a mutex calls **enter**, the mutex's entry count is incremented. The owner of a mutex can call its member function **leave** to decrement the entry count. When the entry count becomes zero, ownership is released and a blocked thread can gain ownership.

Example

```
VMutex* pmtxGuard;
BOOL bMutexOK;

init()
{
   pmtxGuard = new VMutex();
   bMutexOK = (pmxtGuard!=NULL) && (pmtxGuard->getHandle()!=0);
}

someFunction( )
{
```

```
    pmtxGuard->enter();

    // code guaurded against simultaneous execution by multiple
    // threads is placed here

    pmtxGuard->leave();
}

terminate()
{
    delete pmtxGuard;
}
```

class VId

classs **VId** abtracts synchronization IDs, as implemented in VMM services **_BlockOnID** and **_SignalID.** The implementation is found in **INCLUDE\VID.H**.

Member Functions

Block	Block the calling thread
`Signal`	Signal to release all threads blocked on this object

class VGlobalEvent

A global event is the least restricted of the event classes. The VMM can invoke the handler of a global event when all outstanding interrupt processing is complete, regardless of which virtual machine is active.

Most VMM services cannot be reentered; therefore, it is not safe to call them while processing hardware interrupts. However, it is safe to allocate and schedule events inside a hardware interrupt handler. The event's handler will be called once it is safe to call all VMM services. The VMM processes events when making the transition from ring 0 to ring 3.

Member Functions

VGlobalEvent	Constructor
schedule	Arrange to have handler called back
call	Schedule invocation of handler or call handler immediately
cancel	Cancel previously scheduled invocation of handler
handler	Function to be invoked, i.e., the event handler

Using class VGlobalEvent

You use a class derived from **VGlobalEvent** if you need to perform operations related to a hardware interrupt that involve non-async VMM calls. Here are the steps:

- Derive a class from **VGlobalEvent**. Your class must define a constructor and override member function **handler**.
- Create a constructor for your class that invokes the base class constructor, passing to it the parameters that define your global event. The only parameter is the reference data value, and it is optional.
- Write the event handler for your class. The arguments passed to the handler are the handle of the current virtual machine, a pointer to the virtual machine's client register structure, and the reference data value passed to the constructor.
- In the initialization code for your VxD, call **VEvent::initEvents**.
- In your interrupt handler, create an instance of your event class using **operator new**. Base class **VEvent** allows allocation at interrupt level.
- After you create an instance of your class, you can call either member function **call** or member function **schedule**, both of which will eventually result in the invocation of member function **handler**.

Example

The following code goes in your include file:

```
class MyGlobalEvent : public VGlobalEvent

{

MyGlobalEvent();

    virtual VOID handler(VMHANDLE hVM,

CLIENT_STRUCT* pRegs,

          DWORD refData);

};
```

The following code goes in your class implementation file:

```
MyGlobalEvent::MyGlobalEvent() : VGlobalEvent() {}


VOID MyGlobalEvent::handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs,
          DWORD refData)
{
// whatever your handler does
// (now safe to use any VMM service)
}
```

The following code goes in the initialization code for your VxD:

```
if (!VEvent::initEvents())
// fatal error
```

The following code goes in your interrupt handler:

```
MyGlobalEvent* pEvent = new MyGlobalEvent();
if (pEvent)
pEvent->schedule();
```

Notes

Instances of class **VGlobalEvent** are automatically deallocated when the handler returns.

class VPriorityVMEvent

Class **VPriorityVMEvent** allows you greater flexibility in defining the conditions under which an event handler is to be invoked. Similarly to **VVMEvent**, you specify a virtual machine in the call to the constructor. The VMM calls the event handler when the system is/ in the context of the specified virtual machine.

The class constructor allows you to specify a priority boost to apply the specified virtual machine. The priority boost is automatically removed after the event handler runs. Passing a negative boost value lowers the virtual machine's priority.

The class constructor also allows you to specify conditions for the invocation of the event handler, for example, the enabling of interrupts in the virtual machine.

Member Functions

VPriorityVMEvent	Constructor
call	Schedule invocation of handler or call immediately
cancel	Cancel previously scheduled invocation of handler
handler	Function to be invoked, i.e., the event handler

Using class VPriorityVMEvent

You use a class derived from **VPriorityVMEvent** if you need to perform operations in the context of a virtual machine that might not be the current virtual machine. Here are the steps.

1. Derive a class from **VPriorityVMEvent**. Your class must define a constructor and override member function **handler**.
2. Create a constructor for your class that invokes the base class constructor, passing to it the parameters that define your global event.
3. Write the event handler for your class. The arguments passed to the handler are the handle of the current virtual machine, a pointer to the virtual machine's client register structure, the reference data value passed to the constructor, and a Boolean that indicates whether or not the VMM was able to invoke the event handler within the time-out period specified in the call to the constructor.
4. In the initialization code for your VxD, call **VEvent::initEvents**.
5. Create an instance of your event class using **operator new**. Base class **VEvent** allows allocation at interrupt level.
6. After you create an instance of your class, you call member function **call**, which will eventually result in the invocation of member function **handler**.

Example

The following code goes in your include file.

```
class MyVMEvent : public VPriorityVMEvent
{
public:
    MyVMEvent(VMHANDLE hVM);
         virtual VOID handler(VMHANDLE hVM,
    CLIENT_STRUCT* pRegs, DWORD refData, BOOL bTimeOut);
};
```

The following code goes in your class implementation file.

```
MyVMEvent::MyVMEvent(VMHANDLE hVM) :
    VPriorityVMEvent(hVM, LOW_PRI_DEVICE_BOOST, 0, 0,
            PEF_ALWAYS_SCHEDULE) {}

VOID MyVMEvent::handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs,
            DWORD refData, BOOL bTimeOut)
{
    // whatever your handler does
    // (now running in the context of specified VM)
}
```

The following code goes in the initialization code for your VxD.

```
if (!VEvent::initEvents())
    // fatal error
```

The following code goes in the code where you wish to schedule the event.

```
MyVMEvent* pEvent = new MyVMEvent();
if (pEvent)
    pEvent->call();
```

class VThreadEvent

A Thread Event is an event whose execution requires the system to be running in the context of a specific thread. Furthermore, a Thread Event resembles a global event and a VM Event in that the VMM will not invoke the event handler until interrupt processing is complete.

Member Functions

VThreadEvent	Constructor
schedule	Arrange to have handler called back
cancel	Cancel previously scheduled invocation of handler
handler	Function to be invoked, i.e., the event handler

Using class VThreadEvent

You use a class derived from **VThreadEvent** if you need to perform operations in the context of a thread that might not be the current thread. Here are the steps:

1. Derive a class from **VThreadEvent**. Your class must define a constructor and override member function **handler**.
2. Create a constructor for your class that invokes the base class constructor, passing to it the parameters that define your event. The parameters are the handle of the thread in which the event should occur and the reference data value, which is optional.
3. Write the event handler for your class. The arguments passed to the handler are the handle of the current thread, the handle of the current virtual machine, a pointer to the virtual machine's client register structure, and the reference data value passed to the constructor.
4. In the initialization code for your VxD, call **VEvent::initEvents**.
5. Create an instance of your event class using **operator new**. Base class **VEvent** allows allocation at interrupt level. After you create an instance of your class, you can call member function **schedule**, which will eventually result in the invocation of member function **handler**.

Example

The following code goes in your include file.

```
class MyThreadEvent : public VThreadEvent
{
   MyThreadEvent(THREADHANDLE);
        virtual VOID handler(THREADHANDLE hThread, VMHANDLE hVM,
        CLIENT_STRUCT* pRegs, PVOID refData);
};
```

The following code goes in your class implementation file.

```
MyThreadEvent::MyThreadEvent(THREADHANDLE hThread) :
VThreadEvent(hThread, 0) {}


VOID MyThreadEvent::handler(THREADHANDLE hThread, VMHANDLE hVM,
CLIENT_STRUCT* pRegs,PVOID refData)
{
    // whatever your handler does
    // (now running in the context of specified VM)
}
```

The following code goes in the initialization code for your VxD.

```
if (!VEvent::initEvents())
    // fatal error
```

The following code goes in the code where you wish to schedule the event.

```
MyThreadEvent* pEvent = new MyThreadEvent();
if (pEvent)
    pEvent->schedule();
```

class VRegistryKey

VTOOLSD provides class **VRegistryKey** to help you work with the Registry in Win32 environments. The Registry is a hierarchical database of system information that is organized in a tree structure. Each node of the tree is a called a *key*. A key can have any number of subkeys, each of which is a key in its own right. The tree structure of the Registry arises from this relationship.

Each key has a name, which is simply a null terminated string.

The DAA class is KRegistryKey.

A key can optionally have a set of *values*. Each value has a name and associated data. The data of a given value can be either a null terminated string or arbitrary binary data.

In Windows 3.1, a key can have only a single value, which has no name. The anonymous value is always of the string type. Anonymous values are supported in all versions of Windows.

Member Functions

VRegistryKey	Constructor
~VRegistryKey	Destructor
reconstruct	Move the key to one of its subkeys
firstSubkey	Initialize subkey enumeration
nextSubkey	Continue subkey enumeration
firstValue	Initialize enumeration of values
nextValue	value enumeration
setValue	Assign anonymous value
setValue	Assign named value
setSubkeyValue	Assign anonymous value to subkey
getValue	Get anonymous value
getValue	Get named value
getSubkeyValue	Get anonymous value of subkey
removeSubkey	Remove subkey from key
removeValue	Remove value from key
flush	Record changes to key in Registry
lastError	Get error code from last operation
operator=(char*)	Assign anonymous value

Using class VRegistryKey

Each instance of class **VRegistryKey** you construct in your VxD corresponds to a node in the Registry tree. If you construct an instance of **VRegistryKey** that specifies a key that already exists in the Registry, then construction of the instance does not modify the Registry. However, if you construct an instance of **VRegistryKey** that specifies a key that does not already exist, and the value of optional constructor parameter *bCreate* is TRUE, then the construction results in the creation of a new key in the Registry.

The destructor for class **VRegistryKey** does *not* destroy keys in the Registry. The destructor only destroys the **VRegistryKey** instance that references some key in the Registry. If changes were made to the key during the lifetime of the **VRegistryKey** instance, then those changes are effected in the Registry file when the destructor is called. To delete keys from the Registry, see member function **removeSubkey**.

In general, **VRegistryKey** instances should be short-lived objects. Typically, they can be allocated on the stack (as local variables), manipulated within the scope of a function, and implicitly destroyed when the function exits. If you need to descend the Registry hierarchy on a dynamically determined path, you can use member function **reconstruct** to change the key associated with an instance of **VRegistryKey** to one of its subkeys. By doing so, you avoid creating a separate instance of **VRegistryKey** for each node in the path.

You can enumerate all the subkeys of a given key using member functions **firstSubkey** and **nextSubkey**. Similarly, you can enumerate the named values of a key using member functions **firstValue** and **nextValue**.

Example

This example illustrates usage of class **VRegistryKey**. It converts the anonymous value of a particular key to upper case.

```
BOOL valueToUpperCase(VRegistryKey& vkey)
{
        char anonValue[80];
        DWORD cbValue = sizeof(anonValue);

        if (vkey.getValue(anonValue, &cbValue))
        {
                char* p = anonValue;
                while (*p++)
                        if (('a'<=*p) && (*p<='z')) *p &= ~0x20;
                vkey = anonValue;
        }
        return vkey.lastError() == ERROR_SUCCESS;
}

BOOL XRegistryDevice::OnDeviceInit(VMHANDLE hVM, PCHAR
pszCmdTail)
{
        char* path = "SOFTWARE\\Classes\\.classes\\MPlayer";
        VRegistryKey vkey(PRK_LOCAL_MACHINE, path);
```

```
        if (vkey.lastError() == ERROR_SUCCESS)
                if (valueToUpperCase(vkey))
                        dout << "XREGISTR: success";
                else
                        dout << "XREGISTR: error "
<< vkey.lastError();
                dout << endl;
        return TRUE;
}
```

Notes

Of the predefined registry keys supplied by the system, only **PRK_LOCAL_MACHINE** is available to VxDs during initialization. After initialization, the remaining predefined keys (**PRK_CLASSES_ROOT**, **PRK_USERS**, **PRK_CURRENT_USER**) can be used.

The member functions of **VRegistryKey** return **BOOL**. You use member function **lastError** to obtain the precise error code returned by the VMM if a member function returns FALSE.

class VDosToWinPipe

This class, which is derived from **VPipe**, is designed to provide a data channel from a non-system virtual machine to a 16-bit Windows application. The class utilizes the extensible locking and notification mechanisms provided by the base class to implement a pipe that carries a byte stream between the virtual machines.

The locking mechanism uses an embedded instance of class **VSemaphore** to provide exclusive access to non-reentrant portions of the pipe code. When member functions read or write calls **lock** and the pipe is already locked, the caller blocks on the semaphore until **unlock** is called.

If the non-system virtual machine is unable to complete a write request because the pipe is full, it will wait on an embedded instance of **VSemaphore**. After space is made available in the pipe by a read operation, member function **read** calls **OnSpaceAvailable** to signal the semaphore so the write operation can complete.

On the Windows side, VxDs call **read** upon notification of data available while in the context of the system virtual machine. It would be unacceptable to block on a semaphore waiting for data to read, because that would effectively shut down all Windows message processing. Instead, when you create the pipe, you pass the address of an application level call back function to the constructor of **VDosToWinPipe.** When data is available for reading, member function **OnDataAvailable** invokes the call back function using nested execution calls.

Member Functions

VDosToWinPipe	Constructor
write	Write data to pipe
lock	Lock for exclusive access
unlock	Unlock exclusive access
OnDataAvailable	Called when data available for reading
OnSpaceAvailabe	Called when space available for writing
getCallBack	Retrieve application level callback address

Using class VDosToWinPipe

VDosToWinPipe is a useful class "as is", or it can serve as a base class if variations on its behavior are required.

To use this class, you must first identify the source and destination of the data you are transferring, and then provide a mechanism for getting it into and out of your VxD. The pipe provides buffering, exclusive access, and notifications, but it does not define how *applications* specify what data is to be transferred. There are numerous plausible mechanisms by which a VxD would access data to be piped between virtual machines. For example:

- The VxD can provide an entry point by overriding **VDevice::V86_API_Entry** or **VDevice::PM_API_Entry**.
- The VxD can provide an entry point using class **VV86DPMIEntry** and/or **VPMDPMIEntry**.
- The VxD can trap output to a file.
- The VxD can trap output to an I/O port.

Once you have infrastructure to provide your VxD with access to the data it needs to transfer, there is little more to do. From the non-system virtual machine side, simply call member function **write** to put data into the pipe. Keep in mind that this call can block if the pipe is full.

The Windows side of the pipe is only slightly more complicated. Before the pipe can be constructed, a Windows application or DLL must provide your VxD with the *seg16:offset16* address of a function to be called when data is written to an empty pipe. See topic **VDosToWinPipe Callback** in the on-line documentation for a description of this function. This function should call **PostMessage** to notify the application that there is data in the pipe. The application should then call the VxD to read the data.

Notes

Note member functions **lock**, **unlock**, **OnDataAvailable**, and **OnSpaceAvailable** are all protected, and therefore can only be called by other member functions. For any class derived from **VPipe**, member functions **read** and **write** provide the basic public services.

Special Notes for C++ programmers

Constructors for global statics. Care must be exercised in declaring global static instances of classes. Constructors for these classes are called when the VxD receives control message SYS_CRITICAL_INIT. All constructors that use VMM services, either directly or through base class constructors, must ensure those services are safe to call at that time.

It is safer to allocate a global static *pointer* to the class in question, and then allocate the class with operator **new** in **OnDeviceInit**, or at some other appropriate time.

_set_new_handler. VTOOLSD provides limited support of this standard C++ library function. The purpose of the function is to set up a callback function to be called by operator new when a memory allocation request cannot be fulfilled. The class library provides this capability only for the default heap.

7 Programming Topics

Controlling Segmentation

VxD Segmentation

For the Borland compiler, the preprocessor commands to change data segments have no effect. Although the preprocessor commands to set the code segment will work, we recommend exactly one code segment and one data segment per module. See Chapter 5 for instructions on controlling segments with the Borland compiler

VxDs loaded under Windows 3.x can optionally contain several segments. VTOOLSD allows you to precisely control which segment is used for particular code and data. VxDs for Windows 3.x can have any combination of the following segments.

The following table describes the segments VxDs can contain and gives the preprocessor commands to set the desired segment.

Segment	Description
LOCKED_CODE	Specifies the locked protected-mode code segment. This required segment contains the device control procedure, callback procedures, services, and API procedures of the virtual device. Use the following line to switch to the LOCKED_CODE segment: `#include LOCKED_CODE_SEGMENT`
LOCKED_DATA	Specifies the locked protected-mode data segment. This required segment contains the device descriptor block, service table, and any global data for the virtual device. Use the following line to assign data to the LOCKED_DATA segment: `#include LOCKED_DATA_SEGMENT`
PAGEABLE_CODE	For Windows 3.10, this is as an alias for the LOCKED_CODE segment. Windows 3.11 allows segments with pageable attributes. Windows 95 can move pages of PAGEABLE segments to disk when they are not in use. Use the following line to assign data to the PAGEABLE_CODE segment. `#include PAGEABLE_CODE_SEGMENT`

PAGEABLE_DATA	For Windows 3.10, this is as an alias for the LOCKED_DATA segment. Windows for Workgroups 3.11 allows segments with pageable attributes. Windows 95 can move pages of PAGEABLE segments to disk when they are not in use. Use the following line to assign data to the PAGEABLE_DATA segment. `#include PAGEABLE_DATA_SEGMENT`
INIT_CODE	Specifies the protected-mode initialization code segment. This optional segment usually contains the procedures and services that are used only during initialization of the virtual device. The VMM discards this segment after the INIT_COMPLETE message. Use the following line to assign code to the INIT_CODE segment: `#include INIT_CODE_SEGMENT`
INIT_DATA	Specifies the protected-mode initialization data segment. This optional segment usually contains the data used by the initialization procedures and services. The VMM discards this segment after the INIT_COMPLETE message. Use the following line to assign data to the INIT_DATA segment: `#include INIT_DATA_SEGMENT`
STATIC_CODE	Under Windows for Workgroups 3.11 and Windows 95, these segments of a dynamically loaded VxD remain in memory even after the VxD is unloaded. Use the following line to assign code to the STATIC_CODE segment: `#include STATIC_CODE_SEGMENT`
STATIC_DATA	Under Windows for Workgroups 3.11 and Windows 95, these segments of a dynamically loaded VxD remain in memory even after the VxD is unloaded. Use the following line to assign data to the STATIC_DATA segment: `#include STATIC_DATA_SEGMENT`
DEBUG_CODE	Windows 95 loads these segments only when Windows is running under control of a debugger. Use the following line to assign code to the DEBUG_CODE segment: `#include DEBUG_CODE_SEGMENT`
DEBUG_DATA	Windows 95 loads these segments only when Windows is running under control of a debugger. Use the following line to assign data to the DEBUG_DATA segment: `#include DEBUG_DATA_SEGMENT`
REAL_INIT	Specifies the real-mode initialization segment. This optional segment contains the real-mode initialization procedure and data. The VMM calls this procedure before loading the rest of the virtual device, then discards this segment after the procedure returns. VTOOLSD does not support writing REAL_INIT code in C or C++; an example demonstrating how to combine a REAL_INIT segment in assembly language with a C++ VxD is installed into the EXAMPLES\CPP\REALINIT directory.

Guidelines for Locking Code and Data

In order to efficiently use system resources, it is important minimize the amount of code and data that resides in the locked segments. Here are some guidelines to help you determine which code and data areas must be in the locked segment and which can be in pageable segments.

Areas that must be locked:

- Any code that is invoked asynchronously with interrupts disabled, and any data areas accessed by such code.
- The `Declare_Virtual_Device` macro
- The Control Message Dispatcher
- The OnHardwareInt handler of a virtualized IRQ
- Handlers for Asynchronous Timeouts
- Code that disables interrupts

Areas that should be pageable under Windows 95:

- Handlers for virtualized I/O Ports
- Software interrupt handlers
- V86/PM API handlers
- Event handlers, including appy time event handlers
- Config handlers (i.e. Driver, Enumerator, Loader entry points registered with the Configuration Manager)

Areas that should be init segments:

- Handlers for control messages `SYS_CRITICAL_INIT, DEVICE_INIT, INIT_COMPLETE.`
- Any other code used only during initialization

Switching Between Segments

You cannot use this method to switch data segments if you are using the Borland compiler.

The VTOOLSD header files specify the `LOCKED_CODE` and `LOCKED_DATA` segments. Use the `#include` command described above to change the current segment definition for any code and data that appears after the `#include` statement. You can freely switch back and forth between segments in a single source file; however, arranging your code and data to require fewer switches between segments will result in faster compilation.

Library Routines are Loaded into Each Segment

In order to maximize flexibility, most VTOOLSD library routines are made available in all segments. The version the linker actually links in depends on the segment the caller is in when a call to the library routine is made. For example, library routines that are called only from your `INIT` code will appear in the `INIT` segment only, and will be discarded after initialization. Library routines that are called only from your `LOCKED` code will appear in the `LOCKED` segment only. Routines that are called from multiple code segments will appear multiple times in the VxD.

All of the VMM and VxD wrappers are provided in this multi-segment fashion, as well as the C Run-Time Library routines. The Class Library and C Framework routines are located in the appropriate segments only.

Implementation Details

In order to implement this dual-segment scheme, VTOOLSD includes header files that redefine the multi-segment routines to correspond to the routine appropriate for the current segment. For example, when the `INIT` segment is activated:

```
#include INIT_CODE_SEGMENT
```

the following definition (among others) occurs:

```
#undef  List_Destroy

#define List_Destroy INIT_List_Destroy
```

When you then invoke the `List_Destroy` function in your code, the C/C++ pre-processor expands the `List_Destroy` token to `INIT_List_Destroy`:

```
List_Destroy(theList);
```

is expanded to

```
INIT_List_Destroy(theList);
```

The linker, in turn, loads the `INIT_List_Destroy` function from the library. This is the function which is defined in the INIT segment.

On the other hand, when the `LOCKED` segment is activated:

```
#include LOCKED_CODE_SEGMENT
```

the function is redefined:

```
#undef  List_Destroy

#define List_Destroy LOCK_List_Destroy
```

When you then invoke the `List_Destroy` function in your code, the C/C++ pre-processor now expands the `List_Destroy` token to `LOCK_List_Destroy`:

```
List_Destroy(theList);
```

becomes

```
LOCK_List_Destroy(theList);
```

This time, the linker selects the library function defined in the LOCKED segment.

Names Change

One important implication is that the debugger will disassemble calls to the VTOOLSD libraries using the LOCK_ , INIT_, etc. names rather than the names that you specify in your source code.

Segmentation in the Class Library

The class library does not use the segmentation directives described in this section. Any member functions your VxD calls are placed in a predetermined segment appropriate for that function.

Member functions are assigned to segments based on their usage and on the VMM services they call. If member function *X* calls VMM service *Y*, and *Y* cannot be called at interrupt level, then *X* is not placed in the locked code segment.

Services Provided by the VMM and Other VxDs

The VTOOLSD libraries and header files provide C interfaces for most, if not all, functions that Microsoft makes available to be called by your VxDs. In fact, as just described, almost every function is made available multiple times, once in each type of segment.

By convention, Microsoft names services that are defined with a “C” calling convention with a leading underscore in the name. Because VTOOLSD provides “C” wrappers for all services, the VTOOLSD header files provide alternate names for these Micrososft services so they can be called without specifying the leading underscore. The VTOOLSD on line help files specify all service names without the leading underscore.

C/C++ Run-Time Library

VTOOLSD includes a specialized version of the C run-time library. The C run-time library provided with the compiler is not suited for VxDs for the following reasons.

- Function names do not carry the required segment information.

- Some run-time library functions require start-up code to perform special initialization. This initialization is not performed when a VxD is loaded.
- The standard run-time libraries assume access to DOS and BIOS functions not available to VxDs.

The VTOOLSD C run-time library provides a broad subset of the standard ANSI C library. In addition to the library functions, VxDs can use functions that the compiler provides as *intrinsic* functions. The compiler generates code for these functions in-line in the current segment.

In the online help files, you will find complete documentation of the functions in the VTOOLSD C run-time library, along with the functions that the compilers make available as intrinsic functions.

Using Assembly Language

The need might arise to write a portion of your code in assembly language. You might need to call some existing code that has an assembly language interface, or, you might want to optimize a particularly performance sensitive area of your VxD.

VTOOLSD fully supports interfacing to assembly language. In fact, a substantial part of the code supplied with VTOOLSD is written in assembly language.

There are two approaches you can take. You can use the compiler's in-line assembly capability, or you can create pure assembly language modules.

Using In-line Assembly within C and C++ Functions

If you are using the Borland C/C++ compiler, you must install TASM32 in order to use in-line assembly. TASM32 is a separate product from the compiler.

To embed assembly language in a C or C++ module, use the `_asm` directive. Here is an example:

```
void (*procAddr)();    // pointer to some function
DWORD procArg1 = 0x80004317;
BYTE procArg2 = 'X';

_asm {
        mov     eax, [procArg1]
        mov     bl, [procArg2]
        call    [procAddr]
}
```

C++ programmers should note that data members of classes must be moved to temporary variables before you can access them from assembly language.

With the Borland compiler, it might not be possible to have in-line assembly when there are multiple code segments.

Creating Assembly Language Modules

To build assembly language modules, include the following lines at the top of the module:

```
.386p
NO_SEGMENTS=1
include VMM.INC
include VSEGMENT.INC
```

The following table shows which assembler is required for each of the supported compilers:

Microsoft C 9.0	Microsoft MASM 6.11c or later
Borland C++ 4.x	Borland TASM (or TASM32) version 4.0 or later

Some of the useful macros that VMM.INC provides are listed in the following table:

VXD_CODE_SEG **VXD_CODE_ENDS**	Marks the start / end of the code segment
VXD_DATA_SEG **VXD_DATA_ENDS**	Marks the start / end of the data segment
VXD_ICODE_SEG **VXD_ICODE_ENDS**	Marks the start / end of the init code segment
VXD_IDATA_SEG **VXD_IDATA_ENDS**	Marks the start / end of the init data segment
VXD_LOCKED_CODE_SEG **VXD_LOCKED_CODE_ENDS**	Marks the start / end of the locked code segment
VXD_LOCKED_DATA_SEG **VXD_LOCKED_DATA_ENDS**	Marks the start / end of the locked data segment
VXD_REAL_INIT_SEG **VXD_REAL_INIT_ENDS**	Marks the start / end of the real mode init segment
VxDCall, VxDJmp, VMMCall	Invokes a VxD service
BeginProc, EndProc	Delimits functions

Calling Assembly Language Functions from C

To call assembly language functions from C, use the procedure declaration features of the assembler to provide the correct linkage. VTOOLSD provides the following assembler macros in `include\cdefns.inc`, which can be used in conjunction with either the Borland or Microsoft compilers:

Macro	Purpose	Parameters
DEFCPROC	Begins a C callable procedure with `__cdecl` calling convention	Procedure name, Argument list (enclosed in angle brackets < >)
ENDCPROC	Ends a C callable procedure started by DEFCPROC	Procedure name
DEFSPROC	Begins a C callable procedure with __stdcall calling convention	Procedure name, Argument list (enclosed in angle brackets < >)
ENDSPROC	Ends a C callable procedure started by DEFSPROC	Procedure name

See **EXAMPLES\C\CALLASM** for examples.

When used with the Microsoft Visual C++ 32-bit compiler, functions called from C or C++, functions should preserve registers EBX, ESI, and EDI. Functions can return values to the C caller in EAX.

While it is possible to write class member functions in assembly language, doing so is not recommended. Instead, try embedding assembly language in member functions using the compiler's in-line assembler.

Calling C Functions from Assembly Language

To successfully call C routines from assembler, you need to know the function type you are calling. VTOOLSD provides the following macros in **INCLUDE\CDEFNS.INC** to make this easier:

Macro	Purpose	Parameters
EXTCFUNC	Declares an external C function with a __cdecl calling convention	Function name
EXTSFUNC	Declares an external C function with the __stdcall calling convention	Function name, Number of argument bytes

By default, the make files provided with VTOOLSD generate functions in the __stdcall format. Most functions in the C run-time library are of type `__cdecl`. The table below describes how the compiler modifies, or "decorates", function names and specifies which side of the call is responsible for popping the arguments off the stack.

Function Type	Name decoration	Arguments popped by
__stdcall	Microsoft: @n appended, where n is the number of bytes pushed as arguments to the function, and underscore prefix added . Borland: none	callee
__cdecl	underscore prefixed to name	caller
member functions	extensive	callee

Using VPICD Services from C

VPICD_Virtualize_IRQ, like numerous other services provided by the VMM and other VxDs, requires that the caller provide the address of a callback function. When using the VTOOLSD service wrappers, most such services require that callers pass the address of a small block of memory to be used to store the thunk code. The wrapper fills in the thunk block with code that pushes registers on the stack and invokes the C handler. The wrapper passes the address of the thunk to the VMM or VxD in place of the function provided by the caller.

For VPICD_Virtualize_IRQ, things are only slightly different. This service has not one, but five callback addresses. VtoolsD does not require that you pass five thunk block addresses to the service wrapper. Instead, VTOOLSD provides a set of additional calls you use to set up the thunks before calling VPICD_Virtualize_IRQ.

For each of the five notification callbacks used for IRQ virtualization, there is a corresponding VTOOLSD call that initializes the thunk for that callback. The table below summarizes the relationship between notification callbacks and the functions that create their thunks:

Callback	Thunk Initialization Service
`VID_Hw_Int_Proc`	`VPICD_Thunk_HWInt` `VPICD_Thunk_HWInt_Ex`
`VID_Virt_Int_Proc`	`VPICD_Thunk_VirtInt`
`VID_EOI_Proc`	`VPICD_Thunk_EOI`
`VID_Mask_Change_Proc`	`VPICD_Thunk_MaskChange`
`VID_IRET_Proc`	`VPICD_Thunk_Iret`

Here are the full definitions of the thunk initialization services, which are found in **include\vxdsvc.h**:

```
PVOID __stdcall VPICD_Thunk_HWInt(PVPICD_HWInt_HANDLER Callback,
PVPICD_HWInt_THUNK pThunk);


PVOID __stdcall VPICD_Thunk_HWInt_Ex(PVPICD_HWInt_Ex_HANDLER
Callback PVPICD_HWInt_THUNK pThunk);


PVOID __stdcall VPICD_Thunk_VirtInt(PVPICD_VirtInt_HANDLER
Callback, PVPICD_VirtInt_THUNK pThunk);


PVOID __stdcall VPICD_Thunk_EOI(PVPICD_EOI_HANDLER Callback,
PVPICD_EOI_THUNK pThunk);


PVOID __stdcall VPICD_Thunk_MaskChange(PVPICD_MaskChange_HANDLER
Callback, PVPICD_MaskChange_THUNK pThunk);


PVOID __stdcall VPICD_Thunk_Iret(PVPICD_Iret_HANDLER Callback,
PVPICD_Iret_THUNK pThunk);
```

Each of these functions takes two arguments. The first is the address of the caller's handler, i.e., the notification callback written in C. The second argument is the address of a block of memory where the thunk initialization service will write the thunk. This memory can be statically allocated or dynamically allocated, provided it is locked.

Each of the functions returns an address that is passed to **VPICD_Virtualize_IRQ** inside a **VPICD_IRQ_Descriptor** structure. Note **VID_Hw_Int_Proc** is the only one of the notification callbacks you must provide; for the others, you can set the corresponding field of the **VPICD_IRQ_Descriptor** structure to zero and VPICD will not attempt to make the callback.

You will find the prototypes for the notification handlers in **INCLUDE\VTHUNKS.H**. VxDs can unvirtualize an IRQ using the obscurely named **VPICD_Force_Default_Behavior** service.

For an example, see **EXAMPLES\C\HARDINTC**.

Hooking Device Services

The ability to hook the services of other VxDs enables VxDs to monitor and alter the behavior of the system.

Windows 3.1 does not provide facilities for unhooking these hooks; the description of the **Unhook** services apply to Windows 95 only.

VTOOLSD provides the following calls to facilitate hooking services.

- To hook and unhook a register-based service that is never chained to the previous handler, use

  ```
  PVOID Hook_Device_Service(DWORD Service,PDeviceService_HANDLER
  HookProc,PDeviceService_THUNK pThunk)
  ```

  ```
  BOOL Unhook_Device_Service(DWORD Service,PDeviceService_HANDLER
  HookProc,PDeviceService_THUNK pThunk)
  ```

  The prototype for the handler is:

  ```
  VOID __stdcall Handler(PDSFRAME pDS)
  ```

- To hook and unhook a register based service that can be chained to the previous handler, use

  ```
  PVOID Hook_Device_Service_Ex(DWORD
  Service,PDeviceServiceEx_HANDLER pFunc,PDeviceServiceEx_THUNK
  pThunk)
  ```

  ```
  BOOL Unhook_Device_Service_Ex(DWORD
  Service,PDeviceServiceEx_HANDLER
  HookProc,PDeviceServiceEx_THUNK pThunk)
  ```

  The prototype for the handler is:

  ```
  BOOL __stdcall Handler(PDSFRAME pDS)
  ```

If the handler returns FALSE, then control passes to the previous handler.

- To hook and unhook a C callable service, use

  ```
  PVOID Hook_Device_Service_C(DWORD Service, PVOID pHandler,
  HDSC_Thunk* pThunk)
  ```

  ```
  BOOL Unhook_Device_Service_C(DWORD Service, HDSC_Thunk* pThunk)
  ```

  The prototype for the handler varies depending on the service being hooked, and is therefore of type PVOID. To invoke the previous handler, simply call the address returned by **Hook_Device_Service_C**.

- To invoke the previous handler from inside your device service hook, use

  ```
  VOID Call_Previous_Hook_Proc(PDSFRAME pFrame, PVOID pThunk)
  ```

The parameter that specifies the service for these routines is a constant generated by the VxD service table in which the service is found. VxD service tables are declared in the include files for each system VxD. For example, all of the VMM's services are found inside the VxD service table declared in **INCLUDE\VMM.H**. For a given service, the symbol for the constant that is passed to the hooking and unhooking services is formed by prefixing two underscores to the service name as it appears in the service table declaration. For example, consider VPICD_Virtualize_IRQ. The symbol for the service ID constant is __VPICD_Virtualize_IRQ. Note many services appear with a leading underscore in the service table declaration, and therefore the symbol for the service ID constant has three leading underscores, e.g. ___CopyPageTable.

The pDSFrame parameter passed to the handlers for Hook_Device_Service and Hook_Device_Service_Ex is a pointer to a structure that holds the register values being passed to the service. Any modifications the handler makes to this structure are reflected to the register contents when the handler returns.

The difference between **Hook_Device_Service** and **Hook_Device_Service_Ex** is simply that the handler for the latter returns a BOOL indicating if the service was handled. A return value of TRUE instructs the system to return to the original caller, and a return value of FALSE causes control to pass to the previous handler. This service is only available for Windows 95.

A handler for **Hook_Device_Service** or **Hook_Device_Service_Ex** can invoke the previous handler before returning by calling **Call_Previous_Hook_Proc.** This service is only available for Windows 95.

Hook_Device_Service_C is used to hook services that are C callable. As a rule of thumb, a service is C callable if its name has a leading underscore as it appears in the service table declaration. You can check the DDK documentation to determine definitively if a service accepts its arguments in registers or on the stack, i.e. C style. For an example, see **EXAMPLES\C\HOOKCSVC**.

Using VxDCall and VxDJmp Macros

To call services of VxDs without using a wrapper function, you can use C macros **VxDCall** and **VxDJmp**. The parameter for these macros is the service name as it appears in its VxD service table declaration. The declarations of a given VxD service tables are found in the include file that corresponds to that VxD. Note many service names appear with a leading underscore in the service table. See **EXAMPLES\C\CALLSRV** and **EXAMPLES\CPP\CALLSRVP**.

Debugging VxDs Written in C

Chapter 5 describes how to setup and use a debugger with VxDs built using the VTOOLSD tools. In addition, the C Framework provides several functions which can be used to trace the operation of your VxD and to display useful output.

Three error levels are defined in increasing level of severity: `DBG_TRACE`, `DBG_WARNING`, and `DBG_ERROR`. You can separately control which error level causes a message to be displayed and which error level causes a break into the debugger.

Debugging can be directed to the serial port for remote debugging on a second machine, or to an attached monochrome display.

How to Use the Debug Functions

The debug functions are invoked by macros which are only defined to generate code when the DEBUG symbol is defined at compiler time.

SETDEBUGOUTPUT (dbgOutputDevice)

The **SETDEBUGOUTPUT** macro takes a single argument, which must be one of the following:

DBG_SERIAL	Direct debug output to the attached serial device. Output is performed using the **`Out_Debug_String`** VMM service. The Soft-ICE/W debugger can trap and display messages sent to the **`DBG_SERIAL`** device.
DBG_MONO	Direct debug output to the attached monochrome monitor. Output is performed using the **`Out_Mono_String`** VMM service. You must add the line `DEBUGMONO=TRUE` to the [386enh] section of SYSTEM.INI in order to enable output to the monochrome device.
DBG_MONOCLR	The monochrome monitor is immediately cleared, and output is directed as for **`DBG_MONO`**.
DBG_NULL	Causes debug output to be ignored

SETDEBUGLEVEL (breakLevel, messageLevel)

The **SETDEBUGLEVEL** macro requires two arguments. The first argument specifies the debug level that will cause a break to the debugger. The second argument specifies the debug level that will cause a message to be emitted.

The arguments can be any of the following.

DBG_TRACE	Usually used for tracing program flow and does not indicate an error condition.
DBG_WARNING	Usually used to indicate unusual or notable events and might indicate an error condition that does not result in a fatal situation.
DBG_ERROR	Usually used to indicate an error condition.
DBG_NONE	Used to indicate that none of the DEBUG macros should trigger this behavior.

DEBUGTRACE(char *Message)

If the **DEBUG** symbol is defined at compile time, a debug trace event is issued. If the debug message level is set to **DBG_TRACE, DBG_WARNING**, or **DBG_ERROR**, the *Message* is emitted to the debug device. If the debug break level is set to **DBG_TRACE**, **DBG_WARNING**, or **DBG_ERROR**, a debug interrupt is generated.

DEBUGWARN(char *Message)

If the **DEBUG** symbol is defined at compile time, a debug trace event is issued. If the debug message level is set to **DBG_WARNING** or **DBG_ERROR**, the *Message* is emitted to the debug device. If the debug break level is set to **DBG_WARNING** or **DBG_ERROR**, a debug interrupt is generated.

DEBUGERROR(char *Message)

If the **DEBUG** symbol is defined at compile time, a debug trace event is issued. If the debug message level is set to **DBG_ERROR,** the *Message* is emitted to the debug device. If the debug break level is set to **DBG_ERROR**, a debug interrupt is generated.

Default Values

If the **DEBUG** symbol is not defined during compilation, the debug macros do nothing.

If **DEBUG** is defined, then the following defaults are used:

Output device - DBG_SERIAL

Break level - DBG_ERROR

Message level - DBG_WARNING

Real Mode Initialization

The VxD architecture allows a single 16-bit segment in a VxD. Prior to loading the VxD, the system will call to the start of this segment if it is present. The processor is in real mode (or V86 mode) at the time of the call.

By setting register values on return, the real mode initialization routine can do the following.

- Cause the system to abort loading of the VxD
- Cause the system to abort Windows
- Reserve pages of the V86 address space for use exclusively by the VxD
- Direct the VMM to instance memory regions of the V86 address space across virtual machines
- Set a 32-bit field in the VxD's DDB to a reference data value.

Although VTOOLSD does not support writing this real mode initialization routine in C or C++, you will find a documented example of an assembly language real mode initialization routine in **EXAMPLES\CPP\REALINIT\REALINIT.ASM**.

8 Programming Topics in Windows 95, Windows 98, and Windows Me

Application Time Events

Windows 95 supports a special class of events called *application time events*, more commonly referred to as *appy time events*. Any VxD can request to be called back by the system at a time when it is safe to invoke any Windows application level interface. The event handler for an application time event can make use of special services provided by the SHELL VxD to load and call a ring 3 DLL and to allocate and free ring 3 addressable memory. Application time events are very useful for VxDs that must synchronize with 16-bit Windows applications but cannot be used to interface to Win32 applications.

To understand why appy time events are useful, it is important to understand why there are limitations on what handlers for conventional VM events can do. When a conventional VM event runs in the context of the system VM, there is no synchronization between the VMM's invocation of the event handler and the state of the ring 3 16-bit Windows subsystem. The ring 3 state might be 12 levels deep into a GDI call, or it might not have entered any Windows API at all. This uncertainty, combined with the non-reentrability of the ring 3 16-bit Windows subsystem, makes it unsafe for a conventional VM event handler to invoke arbitrary Windows API entry points via nested execution. Calling Windows APIs asynchronously could corrupt critical data structures that Windows maintains.

Appy time events do not suffer these restrictions because their handlers are not called until the ring 3 16-bit Windows subsystem is in a safe state. Therefore, an appy time event handler is free to call *any* 16-bit Windows API, not just **PostMessage** and **PostAppMessage**. A hidden system helper application synchronizes Windows with the VMM.

To schedule an appy time event, your VxD must call SHELL_CallAtAppyTime. The prototype for that service is:

```
APPY_HANDLE SHELL_CallAtAppyTime(
```

```
APPY_CALLBACK pfnAppyCallBack,
PVOID RefData,
DWORD flags,
DWORD msecTimeout
);
```

This service can be called during a hardware interrupt handler.

The first parameter, `pfnAppyCallBack`, is the address of the event handler. The prototype for the event handler is:

```
VOID _cdecl AppyEventHandler(PVOID RefData, DWORD flags);
```

When your VxD calls SHELL_CallAtAppyTime to schedule the callback, the system associates the value in parameter `RefData` with the event and passes it on the stack when it invokes the handler. The system does not restrict this value; it simply passes it through.

In some cases, your VxD might require notification if the handler cannot be invoked within a specific amount of time from the point when it was requested. Parameters `flags` and `msecTimeout` can be used to set a limit on the amount of time that might elapse before the system calls the handler. Set mask bit CAAFL_TIMEOUT in parameter `flags`, and set `msecTimeout` to the time limit (in milliseconds). If the system is unable to synchronize Windows with the VMM before the specified time elapses, it invokes the event handler with the CAAFL_TIMEOUT bit set in the event handler's parameter `flags`. A VxD that takes advantage of this feature when requesting the event must check the flag when entering the handler; if the timeout bit is set, it is not safe to use appy time services.

Here is the list of special services that an appy time event handler can invoke:

SHELL_LoadLibrary	Load a 16-bit DLL
SHELL_FreeLibrary	Free a 16-bit DLL
SHELL_GetProcAddress	Get address of entry point in 16-bit module
SHELL_CallDll	Call an entry point in a 16-bit module
SHELL_LocalAllocEx	Allocate memory from the ring 3 heap
SHELL_LocalFree	Free memory from ring 3 heap

VTOOLSD includes an example VxD written in C that uses appy time events. You will find it in **EXAMPLES\C\APPYTIME**.

Using Application Time Events with C++

The VTOOLSD class library encapsulates the application time event services in class **VAppyTimeEvent**, which is derived from **VEvent**.

Member Functions for Class VAppyTimeEvent

VAppyTimeEvent	Constructor
schedule	Schedule a callback to the handler
cancel	Cancel a previously scheduled callback
handler	Function to call at application time
CallDLL	static member to invoke DLL entry point from handler
LoadLibrary	static member to load a DLL
FreeLibrary	static member to unload a DLL
GetProcAddress	static member to get address of an entry point in a DLL
IsAppyTimeAvailable	static member to determine if application time event service is available
LocalAlloc	static member to allocate memory from SHELL's ring 3 server's heap
LocalFree	static member to free memory allocated by LocalAlloc

This class follows the normal VTOOLSD pattern for event processing. You must first derive a class from **VAppyTimeEvent**. Your class must provide a constructor and override member function **handler**. At the point in your VxD where you need to schedule a callback synchronized with Windows applications, you create an instance of your event class with operator **new** and call its member function **schedule**. Because **VAppyTimeEvent** is derived from **VEvent**, it is allowable to use operator **new** at interrupt level. When the SHELL services application time events, it calls your handler, passing as an argument the reference data value you supplied in the constructor.

Inside the handler, you can make use of the static member functions that call special services provided by SHELL. Use function **CallDLL** to invoke a function exported by a ring 3 DLL. Use **LocalAlloc** and **LocalFree** to allocate and free ring 3 addressable memory from the heap of a special server application that works in conjunction with SHELL. The memory functions are useful for setting up arguments to the function you invoke with **CallDLL**. After allocating and initializing ring 3 memory, you can include its ring 3 address in the parameters to the ring 3 entry point you are calling.

VTOOLSD includes an example VxD that uses class **VAppyTimeEvent**. You will find it in **EXAMPLES\CPP\APPYTIME**.

Interfacing to Win32 Applications

The Win32 API, originally implemented in Windows NT, is a key feature of Windows 95. Although only a subset of the full API is present, developers have access to a wide range of calls that enable development of 32-bit Windows applications.

The need often arises for a Win32 application to load and communicate with a VxD. For example, you might have an application whose purpose is to display data received from a custom device that requires a VxD. The application will need to load the device driver and set up a channel of communication.

A system VxD named VWIN32 provides services that make this possible. VWIN32 is the ring 0 entity responsible for establishing a calling path from a Win32 application to a VxD. In addition, it enables Win32 applications and VxDs to share synchronization objects such as events and mutexes. VWIN32 also makes it possible for a VxD to make an asynchronous procedure call to a function running in a ring 3 thread.

This section examines some of the techniques you can use to provide communication between your VxD and Win32 applications.

The mechanisms that 16-bit Windows applications use to create linkage to VxD entry points do not work for Win32 applications. Your Win32 application cannot issue an INT 2Fh with AX=1684 and BX=device ID to obtain an entry point. In fact, Win32 applications cannot issue any *software interrupts. Attempting to do so will only crash the application. The same restriction applies to the Vendor Specific Entry Point mechanism, Appy Time Events, and Protected Mode callbacks. VxDs that must communicate with both 16- and 32-bit applications must implement distinct mechanisms for each interface.*

Using DeviceIOControl

The API **DeviceIoControl** is the key to communicating with a VxD from a Win32 application. Here is the prototype for that function

```
BOOL DeviceIoControl(
  HANDLE hDevice,// handle of the device driver
  DWORD dwIoControlCode,// function code
  LPVOID lpInBuffer,// pointer to input data buffer
  DWORD nInBufferSize,// size of input buffer
  LPVOID lpOutBuffer,// pointer to output data buffer
  DWORD nOutBufferSize,// size of output buffer
  LPDWORD lpBytesReturned,  // pointer to output byte count
  LPOVERLAPPED lpOverlapped // ptr to OVERLAPPED structure
};
```

Whenever application code makes a call to **DeviceIoControl**, the VxD corresponding to parameter `hDevice` receives a control message W32_DEVICEIOCONTROL. The call into the VxD is synchronous; i.e., the call invokes VWIN32, which calls the control dispatch entry point of the VxD in the context of the calling thread, not at some later scheduled time.

Now consider the prototype of the handler for control message W32_DEVICEIOCONTROL:

```
DWORD handler(PIOCTLPARAMS pParams);
```

. . . and the definition of the structure that the handler takes as a parameter.

```
typedef struct tagIOCTLParams
{
PCLIENT_STRUCTdioc_pcrs;
VMHANDLE dioc_hvm;
DWORDdioc_VxdDDB;
DWORDdioc_IOCtlCode;
PVOIDdioc_InBuf;
DWORDdioc_cbInBuf;
PVOIDdioc_OutBuf;
DWORDdioc_cbOutBuf;
PDWORDdioc_bytesret;
OVERLAPPED*dioc_ovrlp;
DWORDdioc_hDevice;
DWORDdioc_ppdb;
} IOCTLPARAMS, *PIOCTLPARAMS;
```

The following table shows how the parameters to the ring 3 API DeviceIoControl correspond to the members of structure IOCTLPARAMS.

Parameter	Structure Member
dwIoControlCode	dioc_IOCtlCode
lpInBuffer	dioc_InBuf
nInBufferSize	dioc_cbInBuf
lpOutBuffer	dioc_OutBuf
nOutBufferSize	dioc_cbOutBuf
lpBytesReturned	dioc_bytesret
lpOverlapped	dioc_ovrlp
hDevice	dioc_hDevice

VWIN32 actually passes a pointer to a copy of a portion of the client stack to the control message handler. Note since device handles are specific to a particular process, both the handle and the process identifier in member dioc_ppdb are required to identify a caller.

Obtaining a File Handle to a VxD

In order to obtain a handle to the VxD, the application must call another Win32 API, namely, **CreateFile**, prior to calling **DeviceIoControl** . **CreateFile** is a general purpose file interface, and a special convention is used to indicate that the "file" to be opened is, in this case, actually a VxD. The convention is to prefix the device file name with the string `"\\.\"` (to escape the backslashes correctly in C or C++, this becomes `"\\\\.\\"`). Here is some example code that obtains a handle to a VxD:

```
const char* VxDName = "\\\\.\\MYDRIVER.VXD";
hDevice = CreateFile(VxDName, 0,0,0,
   CREATE_NEW, FILE_FLAG_DELETE_ON_CLOSE, 0);
```

There are several independent cases to consider when using **CreateFile** to obtain a handle to a VxD. It works both for statically and dynamically loaded VxDs. Furthermore, it causes dynamically loadable VxDs to be loaded if they are not already loaded at the time of the call.

There is one important requirement that pertains to all cases. In order for an application to successfully obtain a handle to a VxD, that VxD must supply a handler for W32_DEVICEIOCONTROL, and that handler must return DEVIOCTL_NOERROR (zero) when it receives a call with dioc_IOCtlCode equal to DIOC_OPEN (also zero). If the VxD fails to handle the WDIOC_OPEN call in this way, then **CreateFile** will return an error to the Win32 application instead of a handle that can be used to reference the driver.

In the case where the file named in the call to **CreateFile** is a dynamically loadable VxD, the system first checks to see if the VxD is already loaded. If not, it invokes services of VXDLDR to attempt to load it. If loading succeeds, the system sends control message SYS_DYNAMIC_DEVICE_INIT at this time. The VxD must supply a handler for this control message and return TRUE if it initializes successfully. After initialization, the VxD will receive control message W32_DEVICEIOCONTROL with dioc_IOCtlCode set to DIOC_OPEN.

It is also possible to obtain a handle to a statically loaded VxD (i.e. one loaded during initialization of Windows in response to a registry entry or a device= line in **SYSTEM.INI**), or to a resident VxD that was loaded dynamically. In this case, the name specified in the call to **CreateFile** must not include a file extension (such as ".vxd"), because the system will use the device name of the VxD to locate the installed driver. When a statically loaded VxD is opened with **CreateFile**, the VxD will receive control message W32_DEVICEIOCONTROL with dioc_IOCtlCode equal to DIOC_OPEN, but it will *not* receive SYS_DYNAMIC_DEVICE_INIT.

The system maintains a count of how many handles are open for each VxD. The Win32 API **CloseHandle** causes handles on VxDs to be closed. Whenever a handle to a VxD is closed, the VxD receives control message W32_DEVICEIOCONTROL, with dioc_IOCtlCode set to DIOC_CLOSE (-1).

When the count of open handles becomes zero for a VxD loaded in response to a **CreateFile** call, the system invokes services of VXDLDR to unload the VxD. At this point, the VxD will receive control message SYS_DYNAMIC_DEVICE_EXIT (in addition to the final W32_DEVICEIOCONTROL). The handler for this control message should release any resources it has allocated and return TRUE in order to allow itself to be unloaded. Remember handles are automatically closed when a process exits.

Passing Data Between an Application and a VxD

Once an application has a handle to a VxD, it is easy to pass information back and forth using **DeviceIoControl** calls. The calling application sets structure member dioc_IOCtlCode to the value that identifies the function the VxD is to perform. The function might require additional data, which the application can pass in the buffer pointed to by parameter lpInBuffer. The application indicates the amount of input data by setting parameter nInBufferSize.

Inside the VxD's handler for control message W32_DEVICEIOCONTROL, there is typically a switch statement that branches on the value of dioc_IOCtlCode. By convention, structure member dioc_InBuf points to the input data passed to the VxD. The VxD can use the pointer passed in structure member dioc_OutBuf to store data being returned to the application, up to a limit of dioc_cbOutBuf bytes. It is the responsibility of the VxD to set the location pointed to by structure member dioc_bytesret to the number of bytes actually returned. The VxD and the calling application cooperate on the format of the input and output buffers; the system does not restrict what can be stored there. Many developers reserve the first DWORD of the output buffer for storage of an error/status return value.

Selection of the function code value must take into consideration the conventions specified in the file *winioctl.h*, which is included with the Microsoft Visual C++ compiler. Function code values above 32768 are not affected by these conventions, and are the safest to use.

The handler for W32_DEVICEIOCONTROL returns DEVIOCTL_NOERROR to indicate success. Other values cause the ring 3 API to return FALSE.

Overlapped I/O

VxD designers can often improve system performance by allowing applications to continue execution while an I/O operation is in progress. This technique, called *overlapped I/O*, can free the application to perform other computation while the device is busy transmitting data, thereby improving total system throughout.

The **DeviceIoControl** interface includes support for implementing overlapped I/O. Parameter lpOverlapped is a pointer to a structure defined as follows:

```
typedef struct _OVERLAPPED {
    DWORD O_Internal;
    DWORD O_InternalHigh;
    DWORD O_Offset;
    DWORD O_OffsetHigh;
    HANDLE O_hEvent;
} OVERLAPPED;
```

Before an application initiates an overlapped I/O operation, it allocates a synchronization event by calling the Win32 API **CreateEvent** (set parameter *bManualReset* to TRUE). The application then sets structure member `O_hEvent` to the handle of the event it allocated.

A VxD that supports overlapped I/O tests structure member dioc_ovrlp of the IOCTLPARAMS structure to determine if the calling application is expecting overlapped I/O. If the pointer to the OVERLAPPED structure in that member is not NULL, the handler for control message W32_DEVICEIOCONTROL can initiate the I/O operation and return to the caller before the operation completes. In this case, the handler must save the pointer to the OVERLAPPED structure and set the function return value to -1 to indicate the operation is in progress.

When the handler for W32_DEVICEIOCONTROL returns -1, **DeviceIoControl** returns FALSE to the calling application. The application can call **GetLastError** to obtain the exact error, which in this case will be ERROR_IO_PENDING. The application can continue processing until it needs to synchronize with the device in question, i.e., until it cannot perform any useful task while the I/O is incomplete. At this time, the application calls **WaitSingleObject**, or one of the other Win32 APIs used to block while waiting for an event to be signaled. The event whose handle the application passes to **WaitSingleObject** is the same event whose handle was written in the OVERLAPPED structure.

After the I/O operation is complete, the VxD can optionally set member O_InternalHigh of the OVERLAPPED structure to the number of bytes it transferred, or to some value that indicates an error. The VxD then calls **VWIN32_DIOCCompletionRoutine**, passing the value in member O_Internal of the OVERLAPPED structure. The parameter is actually a ring 0 handle for the synchronization event that was allocated by the application. Calling **VWIN32_DIOCCompletionRoutine** causes the ring 3 event to be signaled. If the application was blocked on it, it will then become unblocked. The application can check the value of structure member O_InternalHigh to determine the status of the overlapped operation.

Event Synchronization

Overlapped I/O is actually a special case of synchronization between ring 0 and ring 3. VWIN32 allows access to the underlying services that make overlapped I/O possible. These services enable applications and VxDs to use event objects in a symmetric fashion. Although it is always the responsibility of the application code to create synchronization events, both the VxD and the application can signal or wait on an event once it has been created.

The following table summarizes the event synchronization functions used by applications and VxDs.

	Application level services (Win32 API)	**VxD level services**
To create an event	CreateEvent OpenVxDHandle	<none>
To signal an event	SetEvent PulseEvent	`VWIN32_SetWin32Event` `VWIN32_PulseWin32Event`
To reset an event	ResetEvent	`VWIN32_ResetWin32Event`
To wait for an event to be signaled	WaitForSingleObject WaitForMultipleObjects WaitForSingleObjectEx WaitForMultipleObjectsEx	`VWIN32_WaitMultipleObject` `s VWIN32_WaitSingleObject`
To close an event	CloseHandle	`VWIN32_CloseVxDHandle`

There are some important rules to remember when using events in VxDs.

- The application must call **OpenVxDHandle** to obtain a ring 0 handle for the event. The VxD cannot use the handle **CreateEvent** returns. The application must pass that handle to **OpenVxDHandle** to obtain an event handle usable at ring 0. **OpenVxDHandle** is an entry point in **KERNEL32.DLL**.
- The VxD can only call the services to set, reset, and wait on events from the context of the system VM, i.e., the current VM must be the system VM. The effect of calling these services at other times is undefined.
- The VxD cannot block on an event during the VMM's event processing cycle. In other words, it cannot call VWIN32_WaitSingleObject in a timeout handler, a thread event handler, a global event handler, a VM event handler, a priority VM event handler, a hardware interrupt handler, or at any other point when no application thread is executing.
- The VxD must call VWIN32_CloseVxDHandle to dispose of the ring 0 handle to the event.

The example in subdirectory **EXAMPLES\C\W32INTF** of your VTOOLSD installation includes an application and a VxD that illustrate some of the design techniques described in this section.

Asynchronous Procedure Calls

Signaling a shared synchronization event is one way for a VxD to communicate with an application. Another means is the Asynchronous Procedure Call, or APC. When a VxD needs to notify an application that something has happened, it can use a VWIN32 service to schedule the invocation of a ring 3 function. The system does not call the ring 3 function immediately; rather, it queues a request to call the function and the call occurs the next time the VMM processes events.

The VWIN32 service in question is VWIN32_QueueUserApc. Its prototype looks like this.

```
BOOL VWIN32_QueueUserApc(PVOID pR3Proc, DWORD Param,
   THREADHANDLE hThread)
```

- Parameter pR3Proc is the address of the ring 3 procedure to call.
- Parameter Param is a parameter which is passed to the ring 3 procedure.
- Parameter hThread is the ring 0 handle of the thread in which the procedure is to run.
- The function returns TRUE if the APC is successfully queued.
- Note a thread context is required. The thread handle parameter is a ring 0 handle, not the ring 3 handle. Passing a ring 3 thread handle to this function will fail. To obtain the thread handle, your VxD can use the VMM service Get_Cur_Thread_Handle when the application calls via DeviceIoControl to supply the function address. Other services that return ring 0 thread handles are Get_Initial_Thread_Handle, Get_Next_Thread_Handle, and Get_Sys_Thread_Handle.

The thread specified in the call to VWIN32_QueueUserApc must be in an alertable wait state in order for the APC to actually occur. To reach this state, it must call one of the following Win32 APIs.

- SleepEx
- WaitForSingleObjectEx
- WaitForMultipleObjectsEx

In each case, parameter *fAlertable* must be set to TRUE.

Invocation of the APC causes the thread in question to return from any of the above APIs. In order to always be available for APC execution, the thread must iteratively invoke one of the above APIs.

VTOOLSD includes an example of a VxD and a companion application that demonstrate some techniques for using APCs. You will find this example in subdirectory **EXAMPLES\C\APC** of your VTOOLSD installation.

Using SHELL_PostMessage

You can get many of the same benefits of APCs by using the service **SHELL_PostMessage**. This is often the simplest and easiest way for a VxD to notify a Windows application (16 or 32 bit) that some event has occurred. This service is not useful for communicating with Win32 Console applications.

The prototype for **SHELL_PostMessage** is:

```
BOOL SHELL_PostMessage( HANDLE hWnd, DWORD uMsg,WORD wParam,
DWORD lParam, PPostMessage_HANDLER pCallback, PVOID refData)
```

The first four parameters are as for the corresponding ring 3 API, PostMessage. The application must have previously provided the VxD with the `hWnd`, presumably via **DeviceIoControl** or other such mechanism.

Parameter pCallback is the address of a function the system calls once the message is posted (this is done asynchronously). Parameter refData is the parameter to be passed to the callback function. If your VxD does not need notification, set these parameters to zero.

Writing I/O Subsystem Drivers

Introduction: the I/O Supervisor

Windows 95 introduces a new architecture for supporting block devices such as CD-ROM, diskettes, and hard drives. The heart of the system is the I/O Supervisor, a system VxD that directs I/O from the file system drivers down to the hardware. The I/O Subsystem is conceptually divided into 32 layers, with the I/O Supervisor at the top and the hardware specific port drivers at the bottom. In between, I/O requests go through various translations and mutations. The I/O Supervisor is responsible for routing I/O requests down through the layers of drivers that comprise the I/O Subsystem.

This infrastructure enables developers to limit the functionality of a driver to the specific purpose for which it is intended. For example, a disk security driver need not be concerned with handling interrupts caused by the drive. Similarly, the functionality of a port driver is limited to supporting the physical characteristics of the drives for which it is responsible; a port driver does not concern itself with how those drives are logically seen on the system. The net result is a more modular, flexible, and easily extensible system. For the developer, the price of these benefits is the requirement to understand the architecture and deal with a formidable array of three letter acronyms.

The on-line help has additional detail on the I/O Subsystem that you will need to complete a driver. At the end of this chapter, there is a list of some of the many abbreviations you will encounter when working with the I/O Supervisor.

From File System to Port Driver

When application code opens a file, the Installable File System Manager (IFSMgr) routes the request to the appropriate File System Driver (FSD). The FSD invokes services of the I/O Supervisor to perform I/O to the logical volumes on which the file system resides. The I/O Supervisor sends the I/O down through a hierarchy of "layer drivers", i.e., the drivers that collectively comprise the hierarchically organized I/O Subsystem.

- At the highest logical level below the file system are the *File System Extension Drivers*. These drivers implement features that enhance the file system, such as encryption or compression.
- In the next layer down are the *Volume Tracking Drivers*. Volume trackers monitor drives that have removable media, and ensure the correct media are loaded when an I/O request is made.
- Below the volume trackers are the *Type Specific Drivers* (TSDs). Each TSD manages a set of devices of the same type, e.g. all CD-ROM devices. TSDs are primarily responsible for determining which physical drives correspond to the logical devices in an I/O request.
- Next come the *SCSIizers*. which are akin to TSDs, but have specific functionality for supporting a class of SCSI devices.
- In the lowest layers are the *Port Drivers*, which are hardware specific. The IOS reserves separate layers for different kinds of devices, such as SCSI, Sockets, ESDI, and NEC floppy. Port drivers deal with I/O addresses, DMA transfers, interrupt handling, and other direct manipulations of the hardware.
- The advantage of this architecture is it allows additional drivers, referred to as *Vendor Supplied Drivers*, to be added in among any of the above-mentioned layers. This makes it possible to provide features and enhancements to the I/O Subsystem in a clean and predictable manner. A driver's layer determines at what point in the I/O processing it has the opportunity to manipulate a request. By selecting a layer judiciously, a VSD selects the appropriate point at which to execute. Sophisticated VSDs can even process the same request at multiple layers.

The Mechanics of an I/O Subsystem Driver

At risk of greatly oversimplifying the workings of the I/O Subsystem, here is the big picture.

- The I/O Supervisor manages a set of data structures called DCBs, which correspond to physical and logical units of various hardware adapters. A Port Driver registers data structures for the adapter(s) it controls and then the I/O Supervisor interrogates the Port Driver to collect the information it needs to create the DCBs associated with each adapter.
- The I/O Supervisor broadcasts the presence of DCBs to all layered drivers by calling a general purpose asynchronous event handling routine that each layered driver must implement.

- Each driver must also provide an entry point to handle I/O requests. When a layered driver is notified of a new DCB, it has the opportunity to add its request handler to the list of handlers that receive I/O requests for the corresponding device. The list of request handlers, referred to as the "calldown list", is associated with the DCB.
- I/O requests, generally originating in File System Drivers, generate data structures called IOPs, which are passed to the I/O request handler of each driver in the calldown list of the DCB named in the IOP.
- If a layered driver can satisfy an I/O request, it does so and returns. If not, it is responsible for passing the IOP to the next request handler in the DCB's calldown list.
- Each IOP contains a callback stack. If a driver wants to be notified when a request completes, the request handler pushes a completion handler entry onto this stack. The completion handler is responsible for popping its entry off the callback stack and calling the next one.
- If the IOP reaches the Port Driver, the Port Driver can choose to begin processing it immediately or to place it on a queue for later processing if the device is busy.

The following sections describe the basic mechanisms of I/O Subsystem. The key items are:

- Coding special data structures for the driver
- Loading the layered driver
- Initializing the driver and registering with the I/O Supervisor
- Handling Asynchronous Event notifications from the I/O Supervisor
- Using Services of the I/O Supervisor via the IOS Linkage Block (ILB)
- Inserting the driver into the calldown list for a device
- Handling I/O requests
- Arranging for notification of when a request is complete

Coding a Layered Driver in C with VTOOLSD

The first key difference between building layered drivers and building other VxDs with VTOOLSD is that instead of declaring the basic data structure with macro **Declare_Virtual_Device**, you use **Declare_Layered_Driver**. This macro declares not only the Device Descriptor Block (required of all VxDs), but also declares a Device Registration Packet (DRP) and an IOS Linkage Block (ILB).

The Device Registration Packet tells the I/O Supervisor about the driver. It supplies the layer number (also known as the load group number, or LGN), the driver revision, and additional information about the capabilities and requirements of the driver. The DRP is used only during initialization. The macro automatically initializes the reference data field of the Device Descriptor Block to point to the driver's DRP.

The IOS Linkage Block is a data structure the I/O Supervisor fills out when the driver registers. The driver uses this data structure to link to services provided by the I/O Supervisor. There is a pointer in the DRP to the ILB, which the **Declare_Layered_Driver** macro sets up automatically.

The macro also declares a forward reference to an Async Event Routine (AER), but does not actually implement an AER. The purpose of the AER is explained later in this chapter. The DRP includes a pointer to the AER, which the macro also sets up automatically .

The macro names each of the entities it declares by prefixing the device name to a predetermined string. The following table specifies shows how the naming convention works for a device named **XDISK**:

Device Registration Packet	**`XDISK_Drp`**
IOS Linkage Block	**`XDISK_Ilb`**
Async Event Routine	**`XDISK_Aer`**

Here are the parameters for macro **Declare_Layered_Driver**, in order:

_dname	Device name, as for `Declare_Virtual_Device`
_lgn	IOS layer group number for driver
_asc	6 character ASCII name to identify driver
_rev	driver revision
_fc	feature code (`DRP_FC_xxxx`, see drp.h)
_if	interface requirements flags (`DRP_IF_xxxx`, see drp.h)

Loading a Layered Driver

The I/O Supervisor loads a Port Driver only if directed to do so by the Configuration Manager. Some devices are detected automatically by the system and the driver name is extracted from the system registry. Port Drivers have the file extension ".PDR", and should reside in subdirectory **SYSTEM\IOSUBSYS** of the Windows 95 installation. Port Drivers are loaded prior to other layered block device drivers.

A Vendor Supplied Driver or other layered block device driver is loaded by simply placing the driver in subdirectory **SYSTEM\IOSUBSYS** of the Windows 95 installation. During initialization of Windows, the I/O supervisor scans this subdirectory for files with the extension ".VXD".

Do not place an entry in **SYSTEM.INI** for a layered block device driver.

Initializing a Layered Driver

After all drivers are loaded into memory, the I/O Supervisor examines the DRP of each one to determine in which layer(s) it resides. IOS initializes layered drivers starting with the Port Drivers in the bottom (layer 31) and ascending to the File System Extension Drivers. Drivers in layer 0 are the last layer to be initialized. Drivers configured for multiple layers receive multiple initialization messages, which are described further below.

A layered driver's control message handler is really just a bootstrap mechanism to set up more direct linkage to the I/O Supervisor. In the first phase of driver initialization, the driver receives control message SYS_DYNAMIC_DEVICE_INIT. This is usually the only control message layered block devices need to handle. When a driver receives this message, it should register with the I/O Supervisor by calling **IOS_Register**. There is exactly one parameter for this service, namely a pointer to the driver's DRP. Upon return from **IOS_Register**, a layered driver checks the field **DRP_reg_result** in the DRP. A value of either DRP_REMAIN_RESIDENT or DRP_MINIMIZE implies success. The control message handler returns TRUE to indicate success.

The second, and more important, phase of driver initialization occurs during the registration call via a message sent to the driver's Async Event Routine, which is discussed in the following section.

Handling Asynchronous Event Notifications

The Asynchronous Event Routine (AER) is the primary means the IOS uses to pass information to a layered driver. The AER is thus analogous to the control message handler used by ordinary VxDs. All messages sent to a driver's AER come from the I/O Supervisor.

Macro **Declare_Layered_Driver** declares a prototype for the AER (as described above) which looks like this:

```
VOID __cdecl DEVNAME_Aer(AEP* pAep);
```

where `DEVNAME` stands for the device name specified in the declaration of the driver.

The parameter to the AER is an Asynchronous Event Packet (AEP), which identifies the type and nature of the event being reported to the driver. Although the AEP structure differs according to the event type, the first field is always an AEPHDR, which is defined as follows:

```
typedef struct AEPHDR {

      USHORTAEP_func;// Function Code
   USHORTAEP_result;// result: zero = no error
   ULONGAEP_ddb;// driver data block Pointer
   UCHARAEP_lgn;// current load group num
   UCHARAEP_align[3];// preserve dword alignment
} AEP;
```

The first field, AEP_func, identifies the event. Examples of important events the I/O Supervisor sends to the driver are the initialization request, notifications of device arrivals and removals, media changes, and notification that the system is shutting down. File AEP.H has reasonable descriptions of most of the events and the corresponding structures.

The first AEP a driver receives is the initialization event, AEP_INITIALIZE. The significance of this call depends on the type of layered driver; for many it is not that critical, but it is of much importance to Port Drivers. It is at this juncture that the Port Driver creates and initializes the data structures used by the I/O Supervisor to represent the devices in the I/O Subsystem.

Port Driver Registration and the IOS Linkage Block

When a Port Driver receives message AEP_INITIALIZE, it creates a data structure called a Device Descriptor Block (DDB), which is distinct from the more familiar DDB common to all VxDs. In the I/O Subsystem, a DDB refers to a data structure that describes an adapter (a.k.a. controller). Although a Port Driver can support multiple DDBs (if it supports multiple adapters), most Port Drivers create a single DDB. A DDB is associated with exactly one layered driver, generally a Port Driver.

In order to create a DDB, a Port Driver must call the I/O Supervisor. This is accomplished using an IOS Linkage Block, or ILB. Each layered driver allocates static non-init storage for an ILB and inserts a pointer to this block in its DRP prior to calling **IOS_Register**. The I/O Supervisor fills in the data structure during registration. Here is the declaration of the structure:

```
typedef struct ILB {
  PFNISP ILB_service_rtn;        // addr of service routine
    PVOID  ILB_dprintf_rtn;        // addr of dprintf routine
    PVOID  ILB_Wait_10th_Sec;      // addr of wait routine
    PVOID  ILB_internal_request;   // addr of request routine
    PVOID  ILB_io_criteria_rtn;    // addr of IOR criteria
routine
    PVOID  ILB_int_io_criteria_rtn;// addr of IOP criteria
routine
    ULONG  ILB_dvt;                // addr of driver's DVT
    ULONG  ILB_ios_mem_virt;       // addr of IOS memory pool
    ULONG  ILB_enqueue_iop;        // addr of enqueue routine
    ULONG  ILB_dequeue_iop;        // addr of dequeue routine
    ULONG  ILB_reserved_1;         // reserved; must be zero
    ULONG  ILB_reserved_2;         // reserved; must be zero
    USHORT ILB_flags;              // flags
    CHAR   ILB_driver_numb;   // number of calls to
AEP_INITIALIZE
    CHAR   ILB_reserved_3;         // reserved; must be zero
} ILB, *PILB;
```

The first field, ILB_service_rtn, contains the address of an entry point in the I/O Supervisor that layered drivers use to invoke a range of important services. In order to request a service, the driver must use yet another structure called an IOS Service Packet, or ISP. The definition of an ISP, and the various requests drivers can make via the ILB, are documented in ISP.H. The IOS service entry has the following prototype:

```
VOID __cdecl ServiceRequestHandler(ISP* pIsp);
```

Layered drivers utilize the ILB at various points, not only during port registration.

To create a DDB, a Port Driver's AER must initialize an ISP, setting field ISP_func to ISP_create_ddb, and then call the IOS service entry point. After this is done, the AER sets field AEP_result in the AEP to AEP_SUCCESS and returns.

The I/O Supervisor, acting on the successful initialization of the driver and the creation of the DDB, initiates an interrogation of the driver by calling the driver's AER with message AEP_DEVICE_INQUIRY. The purpose of this iterative interrogation is to determine the actual devices associated with the Port Driver's adapter(s). (For example, a single IDE adapter can control multiple drives.) Making successive queries, the I/O Supervisor creates a data structure called a Device Control Block, or DCB, for each unit associated with the adapter.

For message AEP_DEVICE_INQUIRY, field AEP_i_d_dcb of the AEP is a pointer to a "fake" DCB used for the purpose of the inquiry. After the I/O Supervisor has collected the information about the devices on the adapter, it issues message AEP_CONFIG_DCB to all layered drivers. This message informs layered drivers of the presence of devices in the system, and is discussed further in the following section.

Inserting a Layered Driver into the Calldown List

When the AER of a layered driver receives message AEP_CONFIG_DCB, it examines the DCB named in field AEP_d_c_dcb of the AEP to determine if the device is one it should monitor. If so, it must call the I/O Supervisor to request that its I/O request handler be inserted into the calldown list for that DCB. By doing so, the driver ensures its I/O request handler will get all I/O requests for the device corresponding to the DCB specified in the AEP.

To make this call, the AER sets up an ISP of type ISP_calldown_insert. The following table indicates the fields of this structure that the AER must initialize before invoking the IOS service routine address (in the ILB):

Field	Initialize to	Comments
ISP_i_cd_hdr.ISP_func	ISP_INSERT_CALLDOWN	function code for ISP
ISP_i_cd_dcb	DCB pointer named in AEP	specifies DCB to insert into
ISP_i_cd_req	Address of driver's request handler	
ISP_i_cd_lgn	Load group number of driver	
ISP_i_cd_flags	Demand flags	
ISP_i_cd_expan_len	IOP expansion size	enables driver to allocate private space in all IOPs

The demand flags (field ISP_i_cd_flags) specify requirements of the layered driver that higher layers must fulfill. Drivers that make no additional demands can simply pass the current value of field DCB_dmd_flags in the DCB. A driver clears any flags corresponding to demands it fulfills.

Handling I/O Requests

The prototype for a layered driver's request handler looks like this.

```
VOID __cdecl RequestHandler(IOP* pIop)
```

The contents of the IOP vary depending on the request. Request handlers determine the function being requested by examining field IOP_ior.IOR_func of the IOP. Note there is an IOR structure embedded in the IOP.

Depending on the nature of the request, the request handler can choose to process it or pass it down the calldown list to next handler, with or without modification.

Before passing the request down the calldown list, the request handler must update field IOP_calldown_ptr in the IOP. This field is a pointer to a structure of type DCB_cd_entry, which is used to build the linked list of request handlers. When a request handler is called, field IOP_calldown_ptr points to the DCB_cd_entry that references the request handler. The request handler finds the pointer to the DCB_cd_entry in the list in field DCB_cd_next, and it must store that pointer in field IOP_calldown_ptr of the IOP. Then it can call the next request handler in the calldown list as follows.

```
// Invoke the next calldown function
((PFNIOP)pIop->IOP_calldown_ptr->DCB_cd_io_address)(pIop);
```

If a request handler passes the IOP down the calldown list, it might arrange to have another entry point in the driver called when the request is processed by a lower layer. Each IOP contains a callback stack (not to be confused with the calldown list, which is associated with a DCB). The purpose of the callback stack is to relay notification that a request is complete to the drivers in the calldown list that require such notification. Each item on the callback stack is a structure of type IOP_callback_entry. Field IOP_callback_ptr of the IOP points to the IOP_callback_entry on the top of the callback stack (this stack grows up). In order to add itself to the callback stack, a request handler sets field IOP_CB_address of the IOP_callback_entry structure pointed to by field IOP_callback_ptr of the IOP to the address of the driver's request completion routine, and then sets the incrementing field IOP_callback_ptr so it points to the next callback stack entry in the IOP. This is done prior to invoking the next handler in the calldown list. Here is some code from the example included with VTOOLSD (**EXAMPLES\C\VSD**).

```
// Add the completion handler for this VSD to the callback stack
   pIop->IOP_callback_ptr->IOP_CB_address =
        VSDXMPL_CompletionHandler;
   pIop->IOP_callback_ptr++;
```

The driver that completes processing the IOP is responsible for initiating the calls up the callback stack. Each completion handler must remove its entry from the callback stack and then invoke the previous entry. Here is some example code from the aforementioned example.

```
// remove top of callback stack
   --pIop->IOP_callback_ptr;
// call previous notification
   pIop->IOP_callback_ptr->IOP_CB_address(pIop);
```

A Summary of Some IOS Terminology

AEP	Async Event Packet	IOS notifies each driver of various device related events by calling the Async Event Routine named in the driver's DRP. The AEP describes the type of event.
DDB	Device Data Block	DDBs correspond to adapters (or, if you prefer, controllers). Each adapter, and therefore each DDB, belongs to a particular layered driver. For each DDB, there can be up to 128 DCBs, which are the "units" of the adapter. For example, there would be a DDB for a communications adapter and, for each of 8 comm ports on the adapter, a physical DCB.
DCB	Device Control Block	DCBs correspond to devices, either physical or logical. The DCB contains information about the type of device and its parameters. When DCBs are created, each driver is given the opportunity (via its AsyncEvent Routine) to insert itself into the calldown chain for that DCB.
DRP	Driver Registration Packet	This structure contains information about a layer driver. It is passed to IOS when the driver registers. Information in the DRP specifies various parameters (e.g., the layer or group number to which the driver belongs), and provides the Async Event Routine address.
ILB	IOS Linkage Block	Each layer driver must allocate (static or on the heap) an ILB for communicating with the IOS. A pointer to the ILB is in the DRP. When the driver registers with IOS, IOS initializes the ILB. The ILB contains addresses of entry points in the IOS that the driver can call.
ISP	I/O Service Packet	A layer driver uses this structure to request services from the IOS. The fields vary depending on which service is being requested.
IOR	I/O Request	This structure contains the parameters for a particular I/O request, (e.g. how many bytes to transfer, start address, etc.). IORs are sent to layer drivers inside IOPs.
IOP	I/O Packet	This structure contains an IOR and additional routing information for a particular I/O request.
TSD	Type Specific Driver	This is a class of layered drivers whose primary responsibility is to translate logical requests to physical requests for a particular set of devices of the same type.
VRP	Volume Registration Packet	Used by IOS clients to declare presence of volumes.
VSD	Vendor Supplied Driver	All layered drivers that do not fall into any other category are referred to as VSDs, and they be present in the I/O Subsystem at various levels.

Notes from Microsoft's DDK

Microsoft's Device Driver Kit includes additional documentation on various specific types of drivers. Space does not permit us to reprint the extensive VCOMM, Plug and Play, NDIS, and IOS documentation provided in the DDK. However, there are several other sections of the DDK documentation which we have extracted because they are of general interest to

device driver developers. Although some of the following tips are written with assembly language programmers in mind, the following information, reprinted verbatim from Microsoft's DDK, offers additional insights in the operation of Windows, along with suggestions for avoiding pitfalls that can affect device driver developers.

Paging Through MS-DOS

When run on hardware for which 32-bit drivers are not available, Windows 95 can be forced to use MS-DOS and/or the BIOS for access to the paging device. When paging through MS-DOS, the VMM changes its behavior in significant ways, and new rules apply to existing VxDs. Windows 95 will also page through MS-DOS if the system is running in safe mode.

Make sure to test your VxD on a configuration which pages through MS-DOS. One way to accomplish this is to go to the Control Panel, and select System, Performance, File System, Troubleshooting, then 'Disable all 32-bit protect-mode disk drivers'.

Pageable VxDs

This section describes how paging works using the Windows 3.1 model. Then, changes to the model for Windows 95 are described.

Under Windows 3.1, VxD code segments are always locked, which means that VxD code always runs non-pre-emptively, with the following exceptions:

- If interrupts are enabled, hardware interrupts might be serviced. Therefore, any data structures that can be accessed by hardware interrupts must be protected by disabling interrupts during the access. (And obviously, any data structures accessed by a hardware interrupt must be locked.)
- Accessing swappable memory might result in the code being pre-empted, even if interrupts are disabled. (The VMM might need to wait for the page to arrive from the swap device.) Therefore, any data structures that exist in swappable memory must be protected by some sort of synchronization mechanism; merely disabling interrupts is not good enough.
- Calling a service that adjusts execution priorities might result in the code being pre-empted if the result of the adjustment is that the current virtual machine no longer has the highest execution priority in the system.
- Calling a service that allocates or frees memory from the system heap or pages from the page allocator might result in the code being pre-empted if the swap file needs to be adjusted to account for the memory being allocated or freed.

- The terms pageable and swappable are synonymous. The VMM uses paging as its form of memory management. It does not swap segments or tasks. Where you see the word swap or a derivative thereof, substitute the corresponding form of the word page.

Windows 95 supports VxD with pageable code segments. While this has the benefits of allowing rarely-used code segments to get paged out, thus freeing up memory, it does come at the cost of adding more rules to follow.

Here are additional rules that apply to Windows 95 pageable VxDs. They are in addition to the existing rules from Windows 3.1.

- Pageable code and data cannot be accessed by a hardware interrupt.
- Code in pageable segments can be pre-empted at any time, even if interrupts are disabled. This rule is so important, I'll say it again. Code in pageable segments can be pre-empted at any time, even if interrupts are disabled. Therefore, any data structures accessed by pageable code must be protected by some sort of synchronization mechanism; merely disabling interrupts is not good enough.
- Here is an example.

```
; THIS CODE IS INCORRECT IF IT IS PAGEABLE!


pushfd              ; Disable interrupts to protect a global
cli                 ; variable, so that the update is atomic
mov     eax, pHead       ; Get the head of the list
mov     ecx, [eax].pNext ; And delete it from the list
mov     pHead, ecx
popfd                    ; End of critical section
```

This code fragment must reside in a locked code segment to be valid; if it resides in a pageable code segment, it is broken. (And if the linked list resides in swappable memory, then it is wrong even if locked.)

The previous example is only the most blatant case of pageable code not handling arbitrary pre-emption. Whenever a data structure is inspected or altered, you must consider the possibility not only of pre-emption, but also of re-entrancy. For example, if you choose to use a semaphore to protect the data structure, you might deadlock if that data structure is accessed by an event callback procedure, called on the thread that currently holds the semaphore.

Calling a service which is not explicitly marked as an asynchronous service can result in paging, and therefore can result in the code being pre-empted.

There is one special case where you can slack on needing to synchronize access to a data structure. Namely, if the data structure is never accessed by an event, a timeout, a hardware interrupt, or any other type of operation that causes code to execute, interrupting other code that might be in progress, and if the data structure is accessed by only one thread in the system, then you might be able to get away with not using synchronization.

If Windows 95 is paging through MS-DOS, allowing VxD code segments to be paged out would be catastrophic. In such situations, the VMM automatically locks all VxD code segments (and VxDLdr does the same for dynamically-loaded VxDs).

Bitness

Some services alter their behavior depending on whether the virtual machine is running in 16-bit or 32-bit protected mode. The VMM determines whether a virtual machine is in 16-bit or 32-bit protected mode by recording whether the virtual machine entered protected mode (via the DPMI services) as a 16-bit or 32-bit application. It does not check whether the current CS is a 16-bit or 32-bit code segment. This means that if a 16-bit DPMI client happens to create a 32-bit code segment and switch to it, the VMM will still treat it like a 16-bit application and use only the low word of the extended registers.

In particular, Win32 programs will appear as 16-bit applications from VMM's point of view. In other words, Win32 programs will not be recognized by VMM as 32-bit applications. This should not be a problem because Win32 programs should be using the DeviceIoControl interface to communicate with VxDs. This is merely a warning not even to try it any other way because it won't work.

Deadlocks

With the greater degree of multi-tasking available in Windows 95, the opportunity for deadlocking the system grows enormously. Moreover, some operations, while not deadlocking the system, effectively shut off multi-tasking until the operation completes. The section on events will discuss various deadlock issues related to events. Here are some other issues:

Remember that there are two components in the system which together control whether a thread can run. VxDs interact with the ring 0 scheduler and time slicer, whose rules for choosing which thread can run can be oversimplified to 'run the highest priority thread not blocked on a ring 0 synchronization object'. Meanwhile, there is also a ring 3 scheduler implemented in Kernel32 which has its own rules for which thread can run, based on things applications do, such as **WaitForSingleObject** or **GetMessage**. In order for a thread to run at ring 3, both schedulers must agree that the thread is runnable.

For example, a common scenario is for a VxD to block thread A at ring 0 and wait until thread B does some work at ring 3. If thread A owns some resource at ring 3 which thread B requires, then the system grinds to a halt because thread B cannot run until thread A releases the resource, but thread A is waiting for thread B to do something.

Another common scenario is for a VxD to attempt to acquire a resource at event or timeout time which the current thread already owns. This results in even shorter deadlock chain, where a thread ends up waiting for itself. Examples of this will be given in the chapter on events, but the general rule is not to block inside an event or timeout. Even if you don't deadlock the system, you will almost certainly cause multi-tasking to halt until the thread unblocks.

Another scenario is to call **Begin_Critical_Section**, followed by some other operation which blocks on a synchronization object. 'Blocking with the critical section' usually deadlocks the system because large numbers of important system operations require the critical section in order to proceed. By holding onto the critical section while waiting for something else, those other important system operations cannot be carried out.

Yet another situation is for thread A to go into a **Resume_Exec** loop, waiting for some operation to be performed by thread B. **Resume_Exec** does not block but merely processes events, so if thread B does not have sufficiently high priority, it will never run and thus thread A will wait forever.

Tips and Traps

This section describes common problems that VxD writers encounter and suggests ways to avoid them.

Tip: To avoid cancelling a timeout after it has been dispatched, ensure that the timeout callback procedure immediately set the variable that holds the timeout handle to zero.

Tip: To avoid cancelling a timeout twice by mistake, use the following method:

```
xor     esi, esi
xchg    [hTimeOut], esi
VMMCall Cancel_Time_Out
```

If this code is executed twice by mistake, the second time will not cause any harm. Note, however, that there is still an opportunity for a race condition to occur between the **xchg** instruction and the call to the **Cancel_Time_Out** service. To be extra sure that you don't cancel the wrong timeout by mistake, put the routine in locked code. If the timeout being cancelled is an asynchronous timeout, you also need to disable interrupts.

Tip: To enumerate all of the threads in the System VM, you can't just call **Get_Initial_Thread_Handle** to retrieve the System VM (or **Get_Sys_Thread_Handle**), and then call **Get_Next_Thread_Handle** until you retrieve the handle of a VM whose parent is not the System VM (or until you get back where you started). The reason is that the initial thread handle happens to be the *last* thread in the list, so the next time you call **Get_Next_Thread_Handle**, you will be bumped into the next VM and think the game is over. Instead, call **Get_Sys_Thread_Handle**, and then call **Get_Next_Thread_Handle** repeatedly until you get back to the system thread handle. For each thread along the way, skip it if the parent VM is not the System VM.

Trap: Forgetting to zero-initialize the thread data slot. Remember that thread data slots are not zero-initialized. When one is allocated, you have to go through every thread in the system and initialize each one.

Traps

The following are all real problems that were encountered by virtual device developers:

Trap: Forgetting to free memory associated with a thread data slot when a thread terminates. When a thread terminates, don't forget to free the contents of the thread data slot while you still can; otherwise, the memory will be leaked.

Trap: Scheduling too many events. The VMM has only a limited amount of space to record events. The space does not grow until the next time the VMM processes events with the critical section free. Scheduling thousands of events in rapid succession will crash the machine. Coalesce duplicate events to reduce the demand on the event heap.

Trap: Allocating too many list elements. As with events, the VMM has a limited amount of space for allocating list elements (unless the list uses the heap, in which case list elements come from the heap and this remark does not apply). Also, as with events, the space for list elements grows only the next time the VMM processes events with the critical section free. Allocating thousands of list elements in rapid succession will crash the machine. If you need thousands of list elements, you should reconsider your data structure.

Trap: Forgetting to pass a priority value in the EAX register to **Call_Restricted_Event** or **Call_Priority_VM_Event**. The result is that the thread or virtual machine gets boosted by a random amount, and the system seems to freeze for extended periods of time. (The debugging version of Windows 95 reports this problem.)

Trap: Calling the registry at unrestricted event time. Unless you set the PEF_WAIT_NOT_NEST_EXEC restriction when scheduling the event, it is possible that the event can be dispatched on a thread that is already in the registry. This deadlocks the system. (The debugging version of Windows 95 reports this problem.)

Trap: Leaving events outstanding when you unload. Dynamically-loaded device drivers must remove all hooks and cancel all events, timeouts, and callbacks before they unload. Otherwise, the hook, event, timeout, or callback causes a jump to an invalid location when the appropriate condition is met. If you need to leave a hook or callback in place (for example, because the unhook failed, or there is no way to cancel the callback), put the hook or callback in a static code segment so that it remains loaded even after your VxD unloads. Of course, the stub in the static code segment shouldn't do anything unless the rest of the VxD is already loaded. (The debugging version of Windows 95 reports many cases of this problem, but not all of them.)

Trap: Looping without blocking. Remember that going into a **Resume_Exec** loop does not actually block the current thread; it merely processes events. If the current thread is the highest-priority thread, nothing happens until the loop ends. For additional discussion of this problem, see Chapter 5, "Events."

Trap: Assuming registers do not change across a call. This typically happens when calling a service whose name begins with an underscore (which in most cases indicates that the service uses the C calling and register convention) and assuming that the EAX, ECX, or EDX registers are be preserved across the call. (The debugging version of Windows 95 often intentionally modifies those registers in an attempt to catch code that relies on this non-feature.)

Trap: Assuming stack parameters do not change across a call. The C calling convention permits the called function to modify the input parameters, so don't rely on their values being preserved across a call. The following example is wrong:

```
push    0               ; flags
push    nPages          ; number of pages to lock
push    page            ; first page to lock
VMMCall _LinPageLock    ; Lock them
; do stuff
VMMCall _LinPageUnlock ; Unlock them
add     esp, 12
```

The **LinPageLock** service is allowed to damage the top three double-words on the stack, resulting in garbage being passed to **LinPageUnlock**. (The debugging version of Windows 95 often intentionally modifies input parameters, in an attempt to catch code that relies on this non-feature.)

Trap: Confusing VMSTAT_PM_Exec with VMSTAT_PM_Use32Mask. When deciding whether to use 16-bit or 32-bit offsets from the client, the rule is that you should use a 32-bit offset if the VMSTAT_PM_Exec bit is set, *and* if one of the VMSTAT_PM_Use32Mask bits is set. If a VxD checks only the VMSTAT_PM_Use32Mask bits, it can end up using 32-bit offsets even though the virtual machine is not in protected mode.

Trap: Leaving files open. If a virtual device opens files, it should close them before giving control back to the virtual machine. Since open files are tracked on a per-VM and per-PSP basis, the application on which you opened the file might create another file or might exit, causing the PSP to change, at which point the file handle becomes invalid. Also, having files open causes the virtual IFS Manager device to have problems when it attempts to transition to the protected-mode file system.

Trap: Changing the DS or ES selectors. The CS, DS, ES, and SS selectors must remain as flat selectors at all times. Do not load any other values into those registers, even temporarily. Doing so will crash the system.

Trap: Setting the direction flag. The direction flag must remain clear (up) whenever control passes to the VMM or another virtual device. You can set the direction flag to the 'down' state, but you must clear the flag before yielding control. (The debugging version of Windows 95 attempts to catch this error.)

Trap: Cancelling events the wrong way. If an event is scheduled as a VM Event, it must be cancelled with the **Cancel_VM_Event** service. Similarly, global events must be cancelled with **Cancel_Global_Event**, thread events must be cancelled with **Cancel_Thread_Event**, and restricted events must be cancelled with **Cancel_Restricted_Events**. Failing to observe these rules results in corrupted memory. (The debugging version of Windows 95 attempts to catch this error.)

Trap: Race conditions between timeouts, events, and hardware interrupts. VxD's often initiate an operation and then schedule a timeout that cancels the operation if it takes too long. A race condition can occur if the timeout is dispatched just as the operation completes. This is particularly true if the user is running high-speed communications software at the same time, because communication interrupts will be streaming in, causing the VxD to schedule and cancel many timeouts. If the VxD cancels a timeout after it has been dispatched, and the system has already recycled the handle and given it to another VxD, . the first VxD would end up cancelling the other VxD's timeout by mistake.

Trap: Assuming that **MapPhysToLinear**(*N*, 1, 0) = **MapPhysToLinear**(0, 1, 0) + *N*. This was never guaranteed, but it happened to be true under Windows 3.1 by accident. This is not true under Windows 95.

Trap: Passing the wrong number of arguments to a service. This happens most often with the **HeapFree** service because the VxD writer forgets that there is a second parameter of flags (which must be zero). In general, VxD writers tend to forget to pass flags to services that accept a bitmask of flags as the last parameter. This results in the service picking up stack garbage as the flags parameter. (The debugging version of Windows 95 attempts to catch this error.)

Trap: Assuming that the high word is always zero. Many VxDs that interface to DLLs at ring 3 pass a function code in the AX register to the VxD's protected-mode entry point, but the VxD looks at the entire EAX register to parse the function code. This code tended to work by accident under Windows 3.1 because the high words of extended registers were usually zero. This is no longer true under Windows 95.

Trap: Forgetting to disable interrupts when hooking an asynchronous service. Remember that asynchronous services can be called at hardware interrupt time. If a hardware interrupt occurs after a hook is installed, but before the pointer is saved in a place where the hook procedure can get to it, the interrupt can result in a call to the service before the hook is ready.

Trap: Using **Begin_Nest_Exec** inappropriately. Any protected-mode procedure called from a nested execution block must be in a nondiscardable segment, must access only data in nondiscardable segments, and cannot switch to a 32-bit stack. (This means, in particular, that only interrupt-safe Windows functions can be called, because anything else might thunk to Win32.) Although these restrictions existed in Windows 3.1 as well, Windows 3.1 almost never spent any time on a 32-bit stack, unless a WINMEM32 or Win32s application was running. But now that Windows 95 has Win32 support built-in, the system spends a large percentage of its time on a 32-bit stack, so the window of opportunity for error is much greater. If you need to call functions that do not respect these restrictions, use an application time event, coupled with the **_SHELL_CallDll** service, passing *lpszDll* = 0 and *lpszProcName* equal to the 16:16 address.

A Class Library Summary

The table presented below summarizes the classes of the VTOOLSD class library.

The on-line documentation provides full descriptions of all classes, including detailed descriptions of each member function.

Framework and Application Interface Classes		
Class Name	**Parent Classes**	**Description**
VDevice vdevice.h	VControlMessageDispatcher	Provides dispatching of control messages, and application entry points.
VVirtualMachine vmachine.h	VControlMessageDispatcher	Provides dispatching of control messages, abstraction of virtual machines, translation of VM handles to objects
VPMDPMIEntry vdpmient.h	VProtModeCallback	Provides alternate application entry point for protected mode applications. Does not require Device ID.
VV86DPMIEntry vdpmient.h	VV86Callback	Provides alternate application entry point for V86 mode applications. Does not require Device ID.
VPMVendorExtension vdpmient.h	VVendorExtension VInChainPMInt	Helper class for VPMDPMIEntry.
VV86VendorExtension vdpmient.h	VVendorExtension VPreChainV86Int	Helper class for VV86DPMIEntry.
VVendorExtension vdpmient.h		Base class for VPMVendorExtension and VV86VendorExtension
VThread	VControlMessageDispatcher	Provides dispatching of control messages, abstraction of threads, translation of thread handles to objects

Interrupt Classes		
Class Name	**Parent Classes**	**Description**
VHardwareInt VSharedHardwareInt vhwint.h		Interfaces to VPICD. Used for virtualization of IRQs.
VInChainPMInt vintrs.h	VInChainInt VProtModeCallback	Used to hook interrupts in protected mode applications. Uses a protected mode callback in the application level handler chain to invoke VxD level handler.
VInChainV86Int vintrs.h	VInChainInt VV86Callback	Used to hook interrupts in V86 mode applications. Uses a V86 callback in the application level handler chain to invoke VxD level handler.
VPreChainV86Int vintrs.h		Used to hook interrupts in V86 mode applications. Handler is called prior to all application level handlers. Also supports call to post-chain handler.
VInChainInt vintrs.h		Base class for VInChainV86Int and VInChainPMInt

DMA Classes		
Class Name	**Parent Classes**	**Description**
VDMAChannel vdma.h		Interfaces to VDMAD. Used for virtualization of a DMA channel.
VDMABuffer vdma.h		Used for abstraction of physically contiguous DMA buffer managed by VDMAD.

Event Classes		
Class Name	**Parent Classes**	**Description**
VGlobalEvent vevent.h	VEvent	Abstracts global event. Scheduled after interrupt processing.
VVMEvent vevent.h	VEvent	Abstracts VM event. Scheduled in context of specific virtual machine.
VPriorityVMEvent vevent.h	VEvent	Abstracts Priority VM event. Scheduled in context of specific virtual machine, and supports additional features.
VThreadEvent vevent.h	VEvent	Abstracts thread event. Scheduled in context of specific thread.
VAppyTimeEvent vappy.h	VEvent	Abstracts application time event.
VEvent vevent.h		Base for event classes.

Fault Classes		
Class Name	**Parent Classes**	**Description**
VProtModeFault vfault.h	VFault	Used to hook faults that occur in protected mode applications.
VNMIEvent *vfault.h*	VFault	Used to hook NMI faults.
VVMMFault vfault.h	VFault	Used to hook faults and interrupts that occur in VxDs.
VV86ModeFault vfault.h	VFault	Used to hook faults that occur in V86 mode.
VFault vfault.h		Base for fault classes
VInvalidPageFault vipf.h		Special hook for page faults not handled by the VMM.

Memory Management Classes		
Class Name	**Parent Classes**	**Description**
VPageObject vmemory.h		Base class for objects to be dynamically allocated on unlocked pages.
VLockedPageObject vmemory.h		Base class for objects to be dynamically allocated on locked pages.
VGlobalV86Area vmemory.h		Base class for objects to be allocated from special heap accessible from V86 code.
VPageBlock vmemory.h		An array of linearly contiguous pages.
VV86Pages vmemory.h		Used to manage access to a range of V86 address space.

Callback Classes		
Class Name	**Parent Classes**	**Description**
VCallback		Base class for VProtModeCallback and VV86Callback
VProtModeCallback	VCallback	Associates VxD handler with address in protected mode application's address space.
VV86Callback	VCallback	Associates VxD handler with address in V86 mode application's address space.

Additional Event Classes		
Class Name	**Parent Classes**	**Description**
VIOPort vport.h		Used to trap access to ports by applications.
VHotKey vhotkey.h		Used to trap pressing of a particular key.
VDeviceAPI vdevapi.h		Used to hook application entry point (PM or V86) of another VxD.

TimeOut Classes		
Class Name	**Parent Classes**	**Description**
VGlobalTimeOut vtimeout.h	VTimeOut	Invoke handler after timed interval elapses.
VVMTimeOut vtimeout.h	VTimeOut	Invoke handler after VM executes for specified time.
VThreadTimeOut	VTimeOut	Invoke handler after thread executes for a specified time.
VAsyncTimeOut	VTimeOut	Like VGlobalTimeOut, but handler runs asynchronously, not at event time.

Debug Classes		
Class Name	**Parent Classes**	**Description**
Vdbistream vdebug.h		Input stream
Vdbostream vdebug.h		Output stream for serial port.
Vmonostream vdebug.h	Vdbostream	Output stream for monochrome display.

VCOMM Classes		
Class Name	**Parent Classes**	**Description**
VCommPortDriver vcomport.h	VDevice	Port driver framework
VCommPort vcomport.h		Abstract comm port

Miscellaneous Classes		
Class Name	**Parent Classes**	**Description**
VList vlist.h		Abstracts VMM's linked list services.
VSemaphore vsemapho.h		Synchronization object based on VMM semaphore.
VMutex vmutex.h		Abstracts VMM mutex services.
VId vid.h		Abstracts "ID" synchronization object.
VPipe vpipe.h		Provides FIFO with extensible I/O and locking mechanisms.
VDosToWinPipe vd2wpipe.h	VPipe	Specialized pipe for transferring data from DOS application to Windows application.
D2WPipeEvent vd2wpipe.h	VVMEvent	Helper class for VDosToWinPipe

B Obtaining a Virtual Device ID

Most VxDs will not require a unique VxD ID. Only VxDs that provide services to other VxDs under Windows 3.x are required to have a unique ID. Virtual devices that completely replace device drivers for standard devices should use the standard ID defined in VMM.H.

In order to avoid conflict in ID numbers, Microsoft maintains a device ID database and allocates IDs upon request.

If your VxD requires a unique ID, send electronic mail to the following Internet address:

VXDID@MICROSOFT.COM

You will receive an automated response that contains an application for an ID. Follow the instructions in the application. Response time recently has been quite good; it should be possible to receive an ID within 1-2 weeks.

A recent test produced the following questionnaire in response:

Contact Name(s):

Phone Number(s):

Alt-Phone Number(s):

Personal Compuserve Acct:

Support Advantage Acct:

Internet Email Address:

Company Name:

Address:

City/State/Zip:

Country:

Company Phone:

Company Fax:

Company Email:

Company CIS Acct:

Number of VxD's planned: __

Number of VxD ID requests enclosed: __

Number of VxD ID's assigned to your company so far: __

(more)

----- Repeat following section for each VxD -----

VxD File Name: (Avoid using a V____D.VXD name if possible.)

Will this VxD be loaded from a TSR? (Yes/No) : ___

Will this VxD call out to an MS-DOS device driver or

TSR using Interrupt 2Fh, Function 1607? (Yes/No) : ___

Please provide the estimated number of API's/Exports below.

V86 functions :

PM functions :

VxD Services :

If this VxD replaces a "standard" VxD, which one does it replace : _____

Summarize the purpose of this VxD :

Please provide a technical summary of this VxD below. (Please include for

example, all interrupts hooked, I/O ports trapped, and memory trapped.) :

In what way or with what products will this VxD be distributed?

Will its API or Services be documented for other companies to call?

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X, Y, Z