Microsoft Windows Network Virtual Device Drivers in PATHWORKS for DOS

Digital's PATHWORKS for DOS version 4.1 personal computer integration software includes two network virtual device drivers for the Microsoft Windows environment. These drivers allow Windows applications operating in a protected processor mode and standard DOS applications in a virtual machine to concurrently access services designed to run in real mode under the DOS operating system. The network virtual device drivers, available only in Microsoft Windows enhanced mode, manage DECnet and NetBIOS operations and permit the full use of these interfaces

By Andrew W. Nourse

Introduction

Microsoft Windows virtual device drivers are loadable software modules that extend the Windows operating system and enable it to support peripheral devices, memory resources, and software applications. Some of these modules allow applications that operate in different processor modes with corresponding differences in memory access to communicate with one another in a network system. Digital's PATHWORKS products make it possible to integrate personal computers into local or wide area network systems. The PATHWORKS for DOS software includes two network virtual device drivers, which manage DECnet and network basic I/O system (NetBIOS) operations in the Microsoft Windows environment for PCs.

This paper begins with a discussion of the Microsoft Windows environment for which the PATHWORKS for DOS product provides network virtual device drivers. The basic processor operating modes and Microsoft Windows operating modes are described, preparatory to an explanation of Microsoft Windows enhanced mode. This explanation is essential because virtual device drivers operate only in enhanced mode.

Next, the paper details the capabilities of virtual device drivers, such as providing the means for Windows and DOS applications to communicate. The focus then turns to the environment for developing Microsoft Windows virtual device drivers and concludes with a description of the structure and functionality of the two network device drivers included in the PATHWORKS for DOS software.

Microsoft Windors Environment

The Microsoft Windows environment is a graphical, multiapplication system for personal computers that use the Intel 80286 or higher microprocessor. For 80286-based systems, the Windows system operates in its standard mode, using the real and protected processor modes. On the 80386 or higher microprocessor, the Windows system can also operate in its enhanced mode, using both protected and virtual processor modes. Enhanced mode allows the Windows system to fully utilize processor features such as virtual memory and multiple virtual machines. Virtual device drivers are available only in this enhanced mode.

Basic Processor Operating Modes

All members of the 80x86 family, including the 80386 microprocessor, calculate addresses in memory by using a segment register and an offset. However, the method for calculating the physical address varies, depending on the processor mode. The basic processor operating modes are real mode, protected mode, and virtual mode.

Real Mode. This mode is used by the DOS operating system exclusively and by most DOS applications. The processor calculates physical addresses by shifting the contents of a 16-bit segment register left by 4 bits and adding a 16-bit offset. Therefore, only the first 1 megabyte (MB) plus 65,519 bytes of a PC's physical memory are directly accessible in this mode.

The basic layout of PC memory is shown in Figure 1. The first megabyte of physical memory is known as conventional memory. This area may include the PATHWORKS implementation of the DECnet transport protocol, namely the DECnet Network Process component, as well as other memory-resident software. In addition, conventional memory may contain the DOS operating system and DOS applications. The next 65,519 bytes are called the high memory area. Bank-switched memory, known as expanded memory, may also be available. In real mode, memory protection and virtual memory are not available, illegal instructions are generally ignored, and I/O instructions are always allowed.

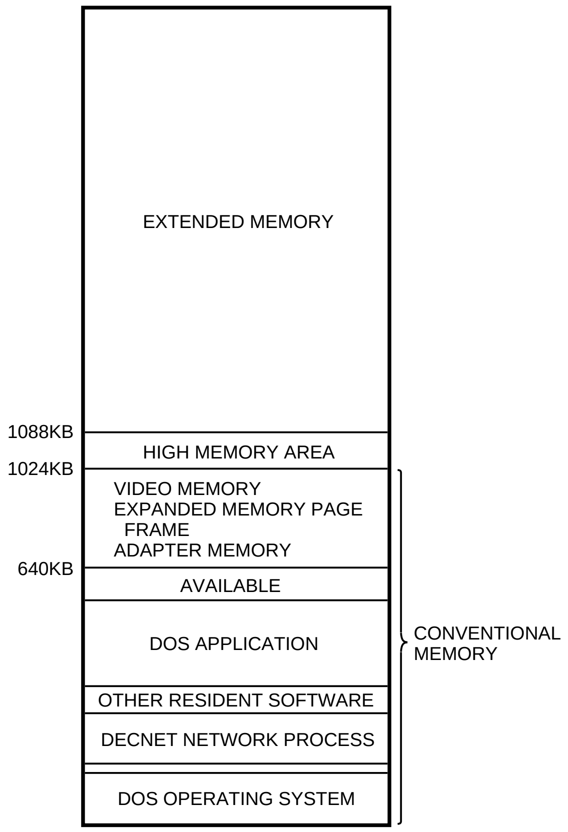

Figure 1 Basic PC Memory Layout

Protected Mode. In this mode, a segment register contains a selector. Part of the selector is an index into a descriptor table maintained by the hardware. A flag in the selector indicates which of two descriptor tables to use, the local descriptor table or the global descriptor table. The processor adds the offset to the linear address obtained from the appropriate descriptor table. The 80386 implementation differs from that of the 80286 because the 80386 processor offers both 16- and 32-bit general registers and offsets, whereas the 80286 processor has 16-bit general registers and offsets.

In protected mode, if paging is disabled, the linear address is the physical address. If paging is enabled, the linear address is decoded into a page directory entry, a page table entry, and an offset. The page directory entry identifies a page table, and the page table entry provides a physical address.

Protected mode is used by applications that use DOS extenders to access memory beyond that which is accessible from real mode. 80386 processors operating in protected mode may use virtual memory. In this mode, an illegal instruction causes a processor trap, and I/O instructions may be selectively allowed or trapped.

Virtual Mode. This mode implements a virtual machine that emulates the behavior of an 8086 microprocessor. Address calculation in this mode is similar to that in real mode, except that in virtual mode the result of the shift-and-add operation is a linear address. The processor converts this address to a physical address, as in protected mode. Processors operating in virtual mode may use virtual memory. Also, each virtual machine can have a separate page directory, an illegal instruction causes a processor trap, and I/O instructions may be allowed or trapped.

Microsoft Windows Operating Modes

The Microsoft Windows environment supports several operating modes.

Windows Real Mode. Similar to previous versions of the Windows system, Windows 3.0 can operate in real mode, i.e., use conventional memory, expanded memory, and the high memory area. This mode is not supported in Windows 3.1.

Windows Standard Mode. Windows 3.0 and 3.1 can operate in standard mode on the 80286 or higher microprocessor. This mode uses the protected processor mode, but does not take advantage of the 32-bit features of the 80386 processor. The Windows system and Windows applications are located outside conventional memory, except for code necessary to provide the communication links with DOS and other resident software. Standard DOS applications run in real mode and occupy the full screen, as if the Windows system were not present. Switching between Windows and non-Windows applications is accomplished by performing a sequence of keystrokes in exactly the same manner as under the MS–DOS version 5.0 task switcher. Virtual device drivers are not used in standard mode.

Windows Enhanced Mode. In enhanced mode, the Microsoft Windows system provides each non-Windows application a virtual machine in which to operate. These machines are preemptively multitasked, so even compute-bound, non-Windows applications can run in the background. The Windows system and all Windows applications share a single virtual machine so they can communicate with each other.

The Microsoft Windows system uses the protected and virtual modes of the 80386 processor. Paging is always enabled. The first 1MB plus 65,519 bytes of the linear address space is mapped to the first 1MB plus 65,519 bytes of memory belonging to the virtual machine currently executing. This mapping allows each DOS application its own block of memory in which to run.

Some data must be shared among the virtual machines. Whenever all or most of the data in a page is shared, a global page is used. Most resident software that was loaded before the Windows system start-up is stored in global pages. Selected data within these global pages may be maintained separately for each virtual machine. This practice is called instancing and may be requested by the resident software.

To support operations requested by virtual machines, virtual device drivers extend the Microsoft Windows kernel. The drivers are loaded at Windows initialization and effectively become part of the kernel.

The Microsoft Windows enhanced mode kernel uses 32-bit registers and offsets. The segment registers are loaded with selectors that allow access to all of memory when the kernel is operating and eliminate the need to break code and data into 64-kilobyte (KB) segments of memory. This memory model is known as the flat model.

Although the Windows enhanced mode kernel is written to use 32-bit registers and offsets, most of the remaining libraries supplied with the Windows system and nearly all applications are written to use 16-bit registers and offsets. The Windows applications run in protected mode, whereas virtual mode provides support for the DOS applications, which need not even be aware that the Windows environment exists.

Virtual Device Driver Capabilities

Virtual device drivers provide the means for Windows and DOS applications to communicate, support asynchronous operations, virtualize hardware ports and interrupts, and directly handle hardware and software interrupts. These capabilities are described in the following section.

Communication between Protected-mode and Real-mode Software Applications

A virtual device driver provides a bridge between Windows applications running in protected mode and DOS terminate and stay resident (TSR) applications written to run in real mode with no knowledge of protected mode. A Windows application that calls an application programming interface (API) passes it a valid protected-mode address. Without virtual device drivers, the real-mode software would interpret this address as a real-mode address, usually pointing to a location within the DOS operating system. A virtual device driver can map the memory into conventional memory and change the addresses so that the real-mode software correctly accesses the caller's data. The virtual device driver should enter a critical section to avoid task switching while calling real-mode software that is not reentrant.

Communication between Transient DOS Application Software and Global Resident DOS Software

Most DOS application software and DOS TSR software is not designed to run in the Microsoft Windows environment. While executing, a DOS application is mapped into conventional memory. If the application calls resident software, and a task switch occurs while an operation is in progress, data would be delivered to the wrong application.

There are two ways to handle this situation. The virtual device driver can enter a critical section to disable task switching until the operation is complete. This approach works well for synchronous operations that never take a perceptibly long time to complete.

However, the system does not respond to most user input while the virtual device driver is in a critical section. Consequently, for long synchronous operations, the end user of the application may believe that the system is hung. If the real-mode software supports asynchronous operations, the virtual device driver can convert the operation to an asynchronous call. Handling the situation in this manner requires that a critical section be entered only for the time it takes to queue the call, and then only if the real-mode software is not reentrant.

Support for Asynchronous Operations

Asynchronous operations, whether in real or protected mode, require that the virtual device driver be able to buffer data in a memory pool that is mapped into every virtual machine. In addition, the driver must set up a completion callback routine to wake up the virtual machine that made the request, deliver the data to that virtual machine, and transfer control to a caller-specified callback routine, if necessary.

Virtualization of Hardware Ports and Interrupts

Another function of virtual device drivers is to virtualize hardware ports and interrupts so that the Windows system can successfully emulate several 8086-based machines at once. Each virtual machine runs a DOS application that assumes it has sole use of a machine. DOS is a minimal operating system and does not provide much of the functionality required by applications. Therefore, most DOS applications bypass the operating system except to access the file system. It is common for an application to set up its own interrupt handlers and to read and write hardware ports. If several applications in separate virtual machines were to attempt these operations at the same time, the applications would interfere with one other. A virtual device driver can trap access to hardware I/O ports and regulate access to the actual hardware.

Direct Handling of Hardware or Software Interrupts

The virtual device driver can provide the functionality of real-mode software. If the user has no need to run this software outside the Windows environment, the software can be removed from memory. Removing the real-mode software reduces the need for context and mode switching, mapping, and copying, and thus offers a considerable performance advantage. If the resident software is removed, more memory is then available to DOS applications running in the Windows environment.

Development Environment

The Microsoft Windows system includes virtual device drivers. Microsoft also has a device driver development kit specifically for developing virtual device drivers.[1] This section describes the environment for developing and debugging this driver software.

Development Tools

Currently, virtual device drivers are written in assembly language because higher-level language compilers generally lack the ability to generate code with 32-bit offsets and registers. A special 32-bit assembler and linker are provided with the Microsoft Windows device driver development kit.

Debugging Tools

Virtual device drivers are debugged using the WDEB386 software module. This debug tool requires that a terminal or equivalent be connected to one of the communication ports on the PC; the debugger performs its I/O to that communications port. Symbols are available in the debugger, but source-level debugging is not provided.

To take full advantage of the WDEB386 capabilities, the debug version of the Microsoft Windows WIN386.EXE module should be used. This version contains many features essential for investigating the behavior of the Windows system and, in particular, for debugging virtual device drivers. The features include commands to display the registers, the stack, and the control blocks for each virtual machine. Many of the virtual device drivers included with the Windows system, and the two included in the PATHWORKS for DOS product, have a debug entry point that may be invoked by entering the period keyboard character, followed by the name of the virtual device driver. Two particularly useful debug entry points are .VMM and .V86MMGR, which provide detailed information about memory usage for each virtual machine, including the use of expanded memory and the high memory area. WDEB386 can be used successfully in the Windows environment to debug virtual device drivers and to diagnose bugs in the read-only memory basic I/O system (ROM BIOS) and other resident real-mode software.

The CodeView for Windows debug tool is intended for debugging applications and dynamic link libraries, not for debugging virtual device drivers. However, the CodeView and WDEB386 tools can be used simultaneously to diagnose problems that occur when applications cause the Windows system to fail.

The Network Virtual Device Drives

The PATHWORKS for DOS software provides two APIs for task-to-task network communications. One is a DECnet socket-based interface, which uses an argument block called an I/O control block (IOCB). The other is the industry-standard PC networking interface, NetBIOS, with some extensions provided by Digital to support wide area networks. The NetBIOS interface uses an argument block called the NetBIOS control block (NCB). Both interfaces are fully supported in Windows enhanced mode.

Digital's PATHWORKS for DOS version 4.1 includes two virtual device drivers to support networking: VDNET.386, which handles DECnet socket calls, and VNETBIOS.386, which handles NetBIOS calls. Although they support different APIs, these two virtual device drivers are similar in structure. The discussion in this section applies to both drivers unless otherwise noted. These drivers are included with the current PATHWORKS version 4.1 product and with Windows version 3.1. To identify Digital Equipment Corporation as the developer of the drivers, Microsoft requested that the module names VDNET.386 and VNETBIOS.386 be changed to DECNET.386 and DECNB.386, respectively, in Windows version 3.1. In this paper, the nomenclature VDNET and VNETBIOS is used to refer to these two modules.

The drivers invoke the real-mode network software in the virtual machine that requested the operation. Creating a "network virtual machine" to which the driver would route all network activity would have allowed most of the network software to be loaded into a single virtual machine and thus freed up conventional memory for non-Windows applications. However, using this design would have incurred the overhead of switching on virtual machines for every network access, timer tick, and network hardware interrupt. In addition, creating a network virtual machine would have required that the data link layer and the DECnet scheduler be capable of performing the virtual machine switch. Finally, this design would be practical only for those users who access the network exclusively while operating in a Microsoft Windows environment.

Initialization

Virtual device drivers are called several times during Windows initialization. While the Windows system is still operating in real mode, the VDNET and VNETBIOS modules check to see if the resident network software is loaded. If it is not, there is no reason to load these drivers. A value is returned that aborts the loading of the drivers but directs the Windows system to continue loading.

After the Windows system enters protected mode, the drivers are called again during each successive phase of initialization. Each virtual device driver takes control of the software interrupts used for its respective API, reserves space in the control block of each virtual machine, obtains parameters from the SYSTEM.INI file, and allocates a pool of global memory for communication with the real-mode resident networking software. Figure 2 illustrates a system virtual machine and a virtual machine running a DOS application. The figure shows the pool of conventional memory that the virtual device driver allocates as global memory.

The drivers perform a "sanity check" to verify that the virtual device driver can distinguish global memory from memory that is local to a single virtual machine. However, the Windows function to perform this check can fail when running on some common unsupported software configurations. At this point, if the sanity check fails, the driver displays a message to advise the user to exit the Windows system.

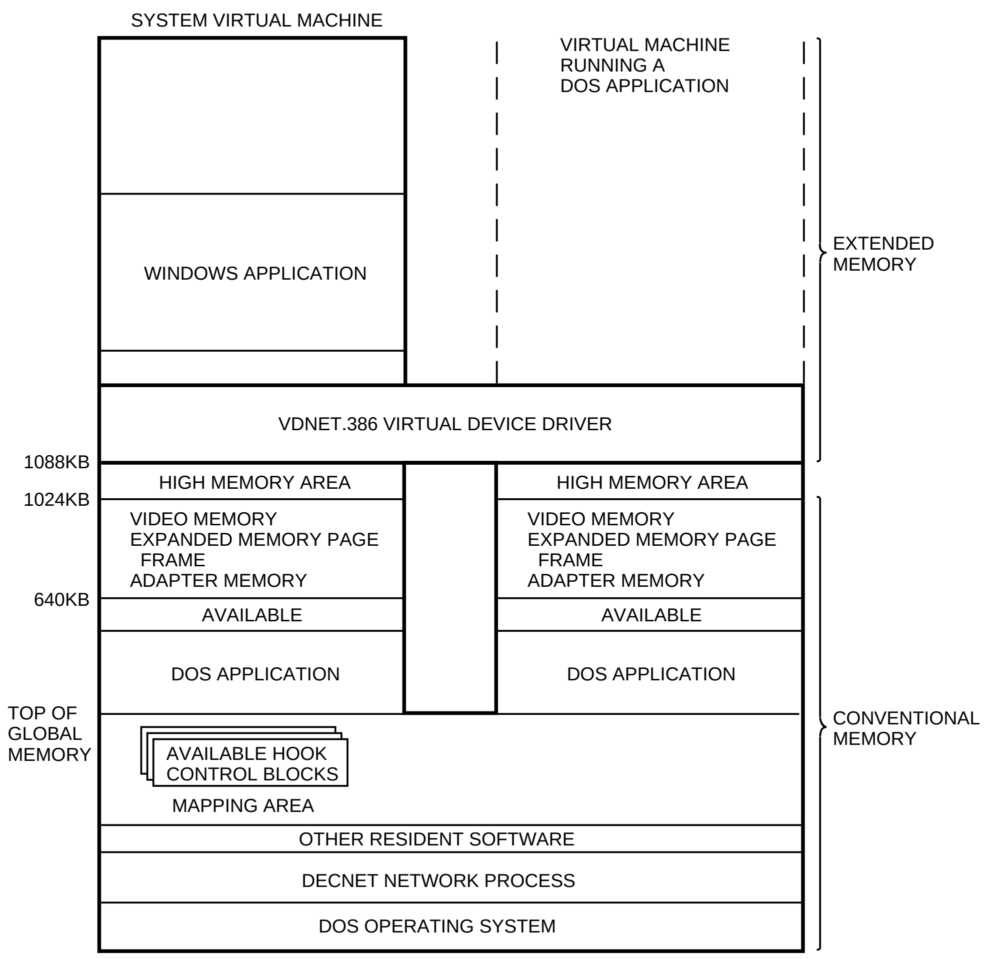

Figure 2 Microsoft Windows Initialization

Virtualization of the Network APIs

When an application issues a software interrupt for a DECnet or NetBIOS call, the appropriate virtual device driver gains control. If the application making the call is in protected mode, the virtual device driver always maps the call in memory. Otherwise, the driver software checks the control block (i.e., the IOCB or the NCB) and the buffer addresses to determine if they are stored in global memory, i.e., mapped identically in every virtual machine. If so, the virtual device driver does not map the call, because it will execute properly without mapping.

API Mapping. If the control block and buffer addresses are not stored in global memory, mapping is necessary. The virtual device driver allocates a hook control block to the operation. This control block resides in global memory and includes an IOCB or NCB, which the virtual device driver passes to the resident networking software. The driver globally maps the caller's buffers in the mapping-space pool allocated at initialization. The IOCB or NCB embedded in the hook control block contains addresses changed to point to the remapped address in the mapping-space pool. The callback (post) address is set to the callback routine in the virtual device driver, so the driver is called when the operation is complete.

Optionally, if the operation is a blocking call that takes a long time to complete, the virtual device driver may convert the operation to an asynchronous call. In this case, the driver sets an internal flag, HF_Suspend_Until_POST, and does not return control to the calling application until the operation is complete. All other virtual machines continue to run while the network I/O is in progress. This design prevents the operation from monopolizing the entire system.

Asynchronous Calls. If the call is asynchronous or has been converted to an asynchronous call, the virtual device driver must establish a callback in order to be notified when the call completes. Because the virtual device driver runs in protected mode and the resident network runs in virtual mode, a special type of callback is required. The virtual device driver uses the Windows Allocate_V86_ Callback service to obtain a real-mode pointer to an instruction in global memory that causes a trap when executed in virtual mode. The Windows system handles this trap and transfers control to the virtual device driver in protected mode.

Invoking the Network Process. The virtual device driver is now prepared to pass the call to the real-mode networking software. The driver enters a critical section to avoid reentrance problems and calls the Simulate_Real-Mode_Interrupt service to invoke the network process as if it were being invoked in real mode. The virtual device driver leaves the critical section when the simulated interrupt returns. If the operation is not asynchronous, the caller's IOCB or NCB is updated, buffers are unmapped, and the hook control block is freed. Figure 3 shows a Microsoft Windows call to the network, intercepted by the virtual device driver and passed to the network process.

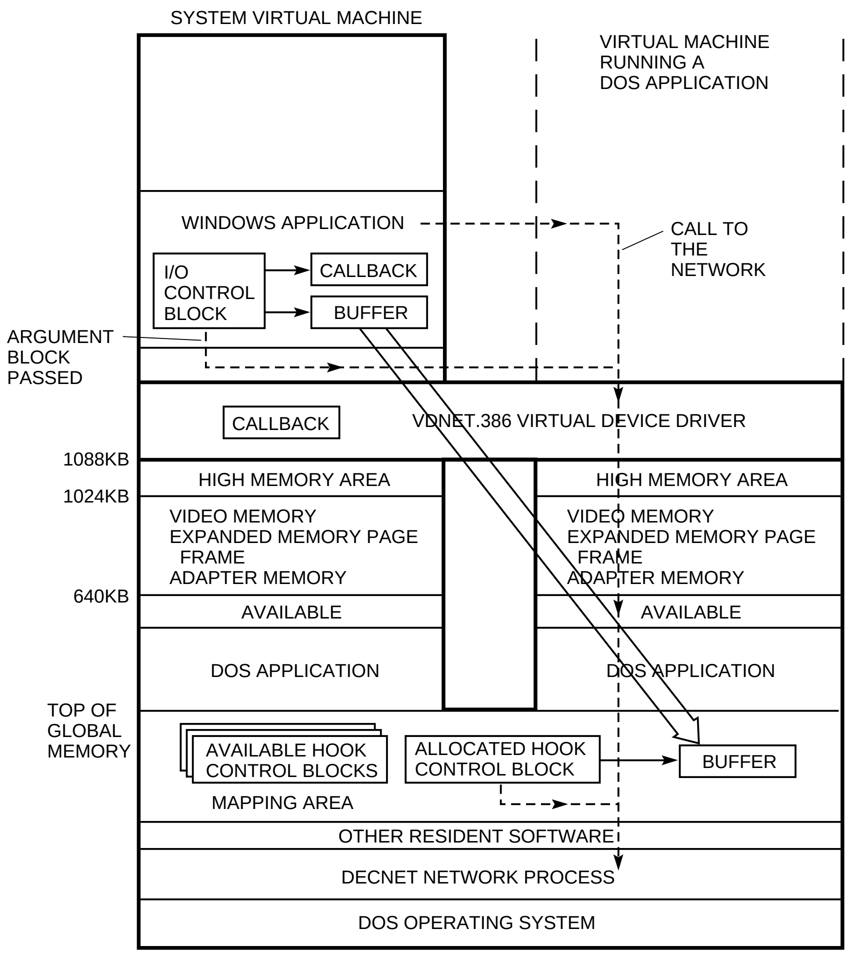

Figure 3 Invoking the Network Process

Callback Routine. The driver checks the HF_Suspend_Until_POST flag to determine if the call was a blocking call that the virtual device driver converted to an asynchronous call. If so, control must not return to the calling application until the operation is complete. Normally, the callback routine in the driver is called at this time. However, certain NetBIOS error conditions cause the operation to return immediately without calling the callback routine. Therefore, the NetBIOS virtual device driver checks the status of the call.

If the call is still in progress, the requesting virtual machine relinquishes its allocated time and retries when the process wakes up. This design protects the process from being awakened prematurely by another virtual device driver. Also, some NetBIOS request errors cause the NetBIOS software interrupt to return immediately and do not transfer control to the callback routine. Ordinarily, the process is only awakened by the callback routine in the virtual device driver on completion of the call.

The Suspend_VM service can be used to block a virtual machine during such a call. However, suspending a virtual machine requires that the system call every Windows virtual device driver to notify it of the suspension. The notification process constitutes a high-overhead operation and is therefore unsuitable for this use.

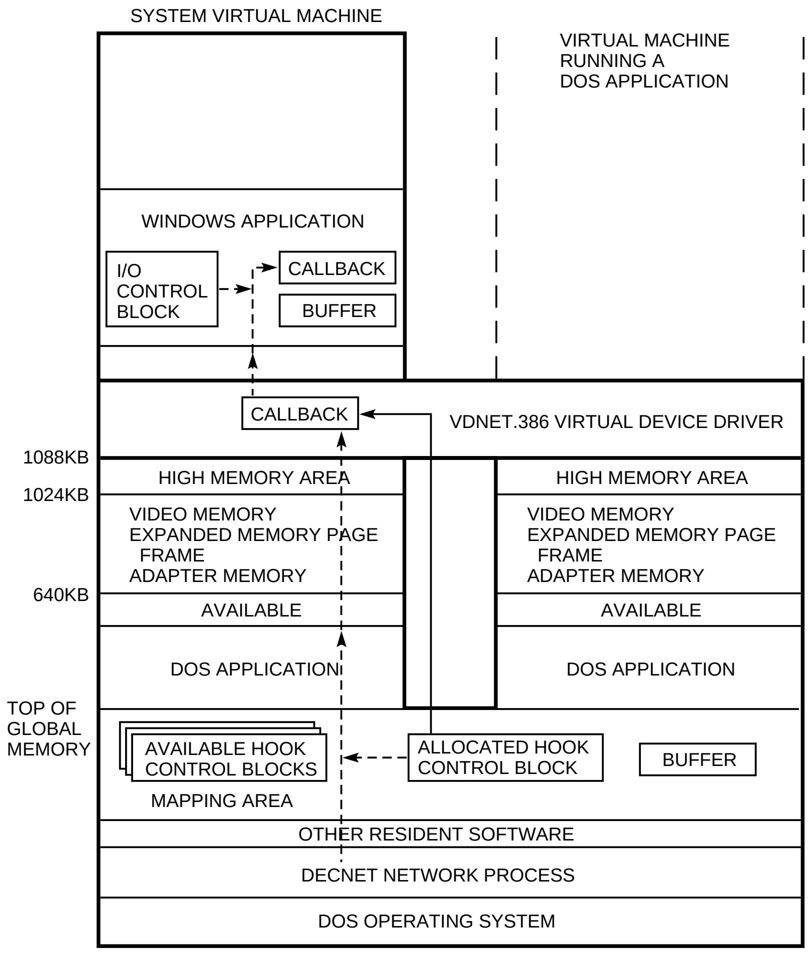

Figure 4 Callback Routine

Table 1
Flags Included in the Debugging Display

Flag	Indication
HF_Wait_For_IRET	Cleared when the DECnet Network Process component returns to the virtual device driver.
HF_Wait_For_POST	Set if the virtual device driver callback is required; cleared when the virtual device driver callback is called.
HF_Wait_For_Sim_POST	Set if the caller requested callback; cleared when the caller's callback returns.
HF_POST_Crit	Set while in a critical section.
HF_From_PM	Set if the caller was in protected mode.
HF_Canceled	Set if the operation was canceled.
HF_Canceling	Set if the operation is being canceled.
HF_Suspend_Until_POST	Set if the operation is a blocking call that is being simulated using an asynchronous call. Do not return to caller until the operation is complete.

If the operation is asynchronous, the system transfers control to the virtual device driver callback routine when the operation is complete, as shown in Figure 4. This routine calls the Windows scheduler to wake up the virtual machine that requested the operation. The Windows event services are also called to invoke the event-handler routine in the virtual device driver when the requesting virtual machine is scheduled. In this way, the virtual device driver regains control. This process restores the caller's context before updating the caller's data.

As shown in Figure 4, the event routine updates the user's argument block and calls the user's callback routine. Finally, the virtual device driver unmaps the buffers, frees up the hook control block, and returns control to the calling application.

Virtual Machine Termination

When a virtual machine terminates, all virtual device drivers are called to perform cleanup. The network virtual device drivers check for outstanding network operations to the virtual machine that is being terminated. All such operations are canceled, and a warning message is displayed to the user. Windows applications execute in the system virtual machine, so their outstanding network operations, if any, are canceled when the user exits from the Windows system. Network operations by resident software are not canceled when a virtual machine terminates.

Debugging Entry Points

The VDNET and VNETBIOS network virtual device drivers provide debugging entry points for use by the Windows kernel debugger. These entry points give a formatted display of the hook control block for each hooked network call in progress. The display includes the requested function, buffer address, the handle of the virtual machine from which the call was requested, the virtual-machine-specific address of the caller's argument block, and flags. The flags included in the debugging display indicate the state of the operation, as shown in Table 1.

Special API Entry Point

The VDNET network virtual device driver provides an API entry point that allows application software to determine what version of the VDNET driver is loaded. This function is available to both protected-mode and real-mode applications.

Summary

PATHWORKS network virtual device drivers extend the Microsoft Windows enhanced mode environment to support most hardware that can be installed in a personal computer. These drivers also support all software that can run under the DOS operating system, including software that bypasses the operating system to access the hardware directly. Network virtual device drivers make network services available to the Windows kernel, to Windows and non-Windows applications, and to other virtual device drivers. The virtual device drivers included in the PATHWORKS for DOS software product permit full use of the DECnet and NetBIOS APIs, including Digital-specific extensions to the NetBIOS interface, in the Microsoft Windows enhanced mode environment.

Reference

1. *Microsoft Windows Device Development Kit—Virtual Device Adaptation Guide* (Redmond, WA: Microsoft Corporation, 1990).

General References

Intel 80386 Hardware Reference Manual (Santa Clara, CA: Intel Corporation, 1987).

Intel 80386 Programmer's Reference Manual (Santa Clara, CA: Intel Corporation, 1987).

Intel 80386 System Writer's Guide (Santa Clara, CA: Intel Corporation, 1987).

Trademarks

Author Biography

Andrew W. Nourse Principal software engineer Andrew Nourse has worked on network software for the PATHWORKS and DECnet–DOS products for the past six years. He developed Microsoft Windows and non-Windows networking applications, libraries, and drivers. Prior to this, he wrote network utilities for DECsystem-20, DECsystem-10, and RSTS/E products. Andy received a B.S. in electrical engineering and computer science from the Massachusetts Institute of Technology in 1974 and joined Digital in 1976.