
Vireo White Paper

Windows™ 95 Interrupt Latency

Karl Heubaum
Sterling Information Group
kheubaum@sterinfo.com

September 27, 1995

Vireo Software presents articles of interest to the development community in the form of Vireo White Papers. For more information about the White Paper series, to request permission to reprint material, or to inquire about submitting an article to be published as a Vireo White Paper, please contact Vireo Software.

Windows™ 95 Interrupt Latency

Karl Heubaum

© 1995 Karl Heubaum. All rights reserved.

The author has taken care in the preparation of this white paper, but makes no expressed or implied warranty of any kind and assumes no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information contained herein.

Microsoft, MS-DOS, Windows, and Win32 are registered trademarks of Microsoft Corporation. Intel and Pentium are registered trademarks of Intel Corporation. Nu-Mega and Soft-ICE/W are trademarks of Nu-Mega Technologies, Inc. VTOOLS and QuickVxD are trademarks of Vireo Software.

The author is a software consultant with Sterling Information Group, 515 Capital of Texas Highway South, Suite 100, Austin, TX 78746.

Introduction

This white paper presents a quantitative analysis of the interrupt latency that can be expected by virtual device drivers (VxDs) executing under the Microsoft® Windows® 95 operating system. A short review of how Windows 95 processes hardware interrupts is followed by a discussion of where latencies are introduced in that process. A technique for measuring latency is then proposed, and the results of applying that technique are presented.

The content of this white paper is derived from a talk I delivered at the 1995 Windows Hardware Engineering Conference (WinHEC) in San Francisco, California on March 22, 1995. The latency calculations included in this white paper have been updated to reflect measurements taken with the release version of Windows 95; the results of those calculations and the conclusions drawn from them are largely unchanged, however, from those presented at WinHEC.

What is interrupt latency?

For the purposes of this paper, “interrupt latency” refers to the time interval that elapses between two events: 1) the instant a hardware device installed in a PC requests service by asserting an interrupt; and 2) the instant the ring 0 device driver code responsible for that hardware begins executing its hardware interrupt service routine. Understanding the length of this interval is important to both hardware and software design. On the hardware side, interrupt latency sets an upper bound for the rate at which a hardware device can generate interrupts and still expect to function — if the device can generate multiple interrupts in the time it takes the interrupt service routine to even begin executing, it is obviously not going to function well. Interrupt latency also dictates buffering requirements for both hardware and software — time spent waiting for latencies is time not spent transferring data, so that data must be buffered somewhere before a transfer can begin. Interrupt latency also affects device driver design — drivers should be designed to take advantage of the minimum latency offered by the operating system without contributing to that latency themselves.

This paper does not discuss the interrupt latency encountered by DLL device drivers executing in ring 3. Placing interrupt service routines in ring 3 code requires Windows 95 to switch from ring 0 to ring 3 with each interrupt, a very expensive operation on Intel® processors; interrupt latencies for this configuration are often an order of magnitude higher than for VxD-based interrupt service routines.

How are interrupts handled by Windows 95?

Hardware interrupt handling under Windows 95 is somewhat complicated. When Windows 95 boots, it initializes the Interrupt Descriptor Table (IDT) so that interrupts are handled by the Virtual Machine Manager (VMM). In the case of hardware interrupts, the VMM cooperates with a VxD called VPICD (Virtual Programmable Interrupt Controller), which virtualizes the dual Intel 8259 interrupt controllers built into IBM-compatible PCs (in today’s machines, dual 8259 functionality is actually built into PCI bus chipsets). When a hardware interrupt occurs, execution transfers to the VMM, which notifies VPICD of the event. VxDs that want to virtualize hardware interrupts register their interest with VPICD, which notifies them once the appropriate interrupt has been asserted and trapped by the VMM. The overall picture looks like this:

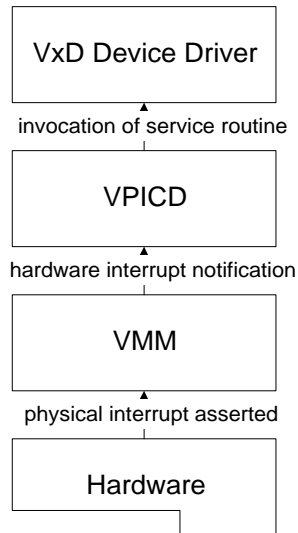


Figure 1

The process of registering to virtualize an interrupt is not trivial, as VPICD offers VxDs a number of different interrupt-related notifications. For the purposes of this paper, the notification of interest is the hardware interrupt notification, which corresponds to the familiar concept of an interrupt service routine.

Where does interrupt latency come from?

Obviously the fact that hardware interrupts are first trapped by the VMM and then passed to VPICD before the actual interrupt service routine is invoked introduces some latency: the VMM is going to execute *X* instructions before invoking an entry point in VPICD, which executes another *Y* instructions before invoking the interrupt service routine. This is a short code path, however, so the latency introduced here is small.

A big contributor to interrupt latency is the interrupt itself. When the 8259 logic interrupts the CPU in response a hardware interrupt, the CPU automatically masks further interrupts (the IF bit in the EFLAGS register is cleared) until the service routine re-enables them (either explicitly by executing the STI instruction or implicitly by returning with an IRET instruction). If another hardware interrupt is asserted while the service routine is still processing the first interrupt, the CPU will ignore it until interrupts are re-enabled, introducing an indeterminate amount of latency. Obviously interrupt service routines need to be carefully designed to do the minimum amount of work necessary to service the interrupt, and the code to perform that work should be optimized for speed. It is possible to explicitly re-enable interrupts in the service routine, but the extra code required to make the routine reentrant often negates the benefit of this approach; moreover, the Windows 95 VMM provides services for scheduling execution of code that can perform more computationally expensive processing outside the context of an interrupt service routine, an approach that provides many of the same benefits (simple service routines that introduce minimal latency) without the complication (reentrancy).

Another potential contributor to interrupt latency is a device that interrupts too often. Regardless of the efficiency of the interrupt service routine, a device that continuously generates interrupts will leave the CPU in the interrupt-disabled mode more often than not. It is also important to remember that software interrupts are interrupts, too. A Win16 or MS-DOS® application that explicitly generates interrupts in order to access operating system services can significantly impact interrupt latency; a MS-DOS program that performs frequent file I/O using the INT 21h services, for example, can increase the interrupt latency observed by hardware devices.

Finally, Windows 95 will explicitly disable interrupts while it is performing operations that would leave the operating system in an unstable state if they were interrupted. A classic example of this situation occurs when the user starts a MS-DOS box, at which time the VMM has to create Local Descriptor Table (LDT) entries, assemble a Task State Segment (TSS), build page table entries, and perform a great deal of other housekeeping that, if interrupted in midstream, could leave the operating system dead in the water. At times like these, Windows 95 will explicitly disable and enable interrupts with the CLI and STI instructions. Unfortunately there are a lot of times like these, and Windows 3.x was justifiably famous for leaving interrupts disabled for long periods of time. The developers of Windows 95 made a conscious effort to alleviate this problem, and they appear to have succeeded.

How do you measure interrupt latency?

It is difficult if not impossible to get an accurate picture of exactly how long interrupt latencies are under Windows 95. It is possible to trace through the VMM and VPICD code paths involved in hardware interrupt processing using a debugger like Nu-Mega™ Technologies' Soft-ICE/W™ and note the instructions that are executed. By counting the cycles required for each instruction on a particular Intel x86 chip and factoring in the chip's clock rate, you could compute an accurate value for that small portion of the overall interrupt latency. Past that point, however, all bets are off. In a real PC, there is an extremely complex interaction of hardware from multiple manufacturers installed on different buses with varying device driver implementation strategies combined with an assortment of 16- and 32-bit Windows applications and MS-DOS programs. Just how all these bits and pieces will combine to affect the latency observed by one interrupt from one hardware device is impossible to predict or calculate. This situation leads to the exchange of war stories among device driver programmers describing why a device failed in a particular customer's PC because it was seeing huge latencies thanks to Manufacturer X's broken hardware or drivers.

One method for obtaining useful — if not absolutely accurate — interrupt latency information involves using high-resolution timers to actually measure latency. A company in Redmond, Washington called Alpha Logic Technologies markets an ISA bus adapter featuring such timers under the name STAT!. It includes five timers, six frequency generators, and an interrupt generator; its minimum resolution is 250ns. A 60MHz Pentium® processor completes one clock cycle approximately every 16ns, so the STAT! board is not useful for timing a few tens of instructions. It can provide useful information, however, for code paths comprising several hundred instructions, and that regime includes hardware interrupt processing and its associated latencies.

Alpha Logic supplies assembler source code for a library of C-callable functions that initialize, configure, start, read, and stop the timers on the board. This source code is intended for use with MS-DOS programs — not VxDs — so I used the VTOOLS™ QuickVxD™ program to generate a skeleton STAT! VxD using the C++ class library framework. I added prototypes for exported services that mirror the original library's entry points, enabled the V86 and protected mode entry points, and added a handler for the Windows 95 W32_DEVICEIOCONTROL message. Once the skeleton VxD was generated, I cut and pasted the original Alpha Logic assembler code into the C++ framework, wrapping it with in-line assembler directives and adding any necessary local variables. I also defined a series of integer message IDs corresponding to the exported services and wrote dispatcher routines driven by those message IDs for the VDevice::OnW32DeviceIoControl(), VDevice::V86_API_Entry(), and VDevice::PM_API_Entry() member functions. Finally, I wrote a V86 mode static library and 16- and 32-bit DLLs that support the same services as the STAT! VxD. The V86 mode library and 16-bit DLL locate the V86 and protected mode VxD entry points, respectively, using DPMI function 1648h and then invoke the appropriate VxD service by passing the corresponding message ID and parameters; the 32-bit DLL works by opening the STAT! VxD with a call to the Win32@ CreateFile() API, then uses DeviceIoControl() to pass the message IDs and parameters.

With mechanisms in place to control the STAT! hardware from VxDs, 16- and 32-bit Windows applications, and MS-DOS boxes, I wrote a second VxD using VTOOLS that actually conducts the latency measurement tests. When instructed by a 32-bit Windows control application (the

communication mechanism here is again DeviceIoControl(), this measurement VxD uses the services exported by the STAT! VxD to initialize the hardware and configure the timers. The input to the timers is a frequency generator with a period of 250ns, and the output is routed to the interrupt generator, which is tied to an ISA bus interrupt line. The timer itself is initialized so it will overflow after 20ms, at which time the interrupt fires, the initial value is reloaded, and the timer begins counting again. The measurement VxD's interrupt handler, derived from the VHardwareInt class, simply reads the timer, stores that value in a static array, and sends an end-of-interrupt to the 8259. After 2,000 measurements have been stored, the measurement VxD disables the timer. The control application then uses DeviceIoControl() to request all of the measurements, which it writes to an ASCII file. That file is then read into a Microsoft Excel spreadsheet, which converts the measurements to elapsed times by subtracting the STAT! timer's initial value from the values returned by the measurement VxD.

This method obviously will not provide exceptionally accurate latency timings, but it does provide reasonable ballpark numbers. Moreover, by varying the conditions under which the measurements are taken, it is possible to explore worst-case scenarios to evaluate how different conditions affect interrupt latency. I evaluated four scenarios in my experiments:

1. Idle.
2. Running four instances of the WINBEZMT.EXE program, each running four drawing threads. WINBEZMT.EXE is a demo application that was distributed with early beta versions of Windows 95. It is a multiple document interface (MDI) application where each document window is paired with a thread that repeatedly draws Bezier curves using the Win32 PolyBezier() function; it is a CPU and GDI intensive application.
3. Running four instances of a MS-DOS program, DOSFP.EXE, that executes an infinite loop, repeatedly performing the floating point division operation used to identify Pentium chips with the infamous FDIV bug.
4. Running four instances of a MS-DOS program, DOSIO.EXE, that executes an infinite loop, repeatedly opening, reading, and closing a 100KB file using a 1KB buffer. Windows 95 quickly caches the file, but DOSIO.EXE generates many software interrupts as it performs file I/O.

All of these measurements reported in this white paper were taken on a Dell OmniPlex 560 PC with a 60MHz Pentium processor and 32MB of DRAM running the release version of Windows 95. These measurements have also been performed on Compaq DeskPro XL PCs with 60 and 66MHz Pentium processors; the results were very similar, with interrupt latencies on the 66MHz machine being very slightly lower, consistent with the higher clock rate. I have no reason to believe the experiment could not be easily repeated on other PCs, although the 250ns resolution of the STAT! board becomes a bigger issue as processors with higher clock rates are tested.

So what interrupt latency can I expect under Windows 95?

The interrupt latency values obtained using this measurement method are summarized in Table 1:

Test Case	Average Interrupt Latency (μ s)
Idle	11
WINBEZMT	21
DOSFP	11
DOSIO	15

Table 1

The 11 μ s interrupt latency value for an idle machine is not at all bad for a non-real-time operating system. As expected, the latency for the DOSFP test case was the same as the idle case – VxD-based

interrupt handlers are invoked in the context of any virtual machine (VM), so the fact that four virtual machines executing compute-intensive infinite loops were running during the DOSFP measurements should not have effected latency. The DOSIO test cases demonstrated a slightly longer latency, which can be attributed to the multitudes of software interrupts that particular MS-DOS program generates as it performs file I/O.

The interrupt latency for the WINBEZMT case is definitely outside the expected range, and deserves additional investigation. Closer examination of WINBEZMT.EXE with Soft-ICE/W reveals that the Bezier curve drawing threads execute a tight loop that repeatedly calls the PolyBezier() function followed by a call to Sleep() with a 1ms delay. Further investigation of the PolyBezier() code path reveals that the PolyBezier() code in GDI32.DLL thunks down to PolyBezier() and PolyBezierTo() in GDI.EXE via the KERNEL32.DLL QT_Thunk mechanism. This is not at all unusual given that most of the GDI functionality in Windows 95 is implemented in 16-bit code in GDI.EXE, and this particular thunking mechanism does not effect interrupts, so the calls to PolyBezier() are probably not the source of the additional interrupt latency.

Further investigation of the calls to Sleep() are a different story. The Sleep() code in KERNEL32.DLL actually thunks down to code in VWIN32.VXD using the INT 30h protected mode callback mechanism. The VMM Allocate_PM_Callback service provides this mechanism so that protected mode components like KERNEL32.DLL can access ring 0 code like VWIN32.VXD. When the INT 30h instruction is executed, interrupts are disabled and execution transfers to the VMM, which examines the address of the INT 30h instruction in order to locate the appropriate ring 0 handler – in this case, code in VWIN32.VXD that actually implements the Sleep() functionality requested by WINBEZMT.EXE. Eventually, of course, interrupts are re-enabled, but this INT 30h protected mode callback mechanism does result in the disabling of interrupts, and with the WINBEZMT.EXE Bezier threads calling Sleep() as fast as they can, there are an extremely high number of interrupts generated. Moral of the story: Even in a system running only 32-bit Windows applications, interrupt latencies can still increase significantly because of otherwise invisible shenanigans.

Also of interest is the maximum interrupt latency encountered for each test scenario:

Test Case	Maximum Interrupt Latency (μ s)
Idle	25
WINBEZMT	46
DOSFP	33
DOSIO	46

Table 2

Table 2 is by no means the final statement on maximum interrupt latencies under Windows 95. As with Windows 3.x, these values will vary from machine to machine and from CPU to CPU. The hardware and device drivers installed in a particular machine will continue to have a potentially huge impact – if the device driver for some manufacturer's hardware decides to compute π to a million digits in the interrupt handler, your hardware and its device driver are going to see long latencies, and there is nothing you can do about it but point the finger and hope your customer accepts that explanation.

Also of interest is the distribution of interrupt latencies, here illustrated as a histogram:

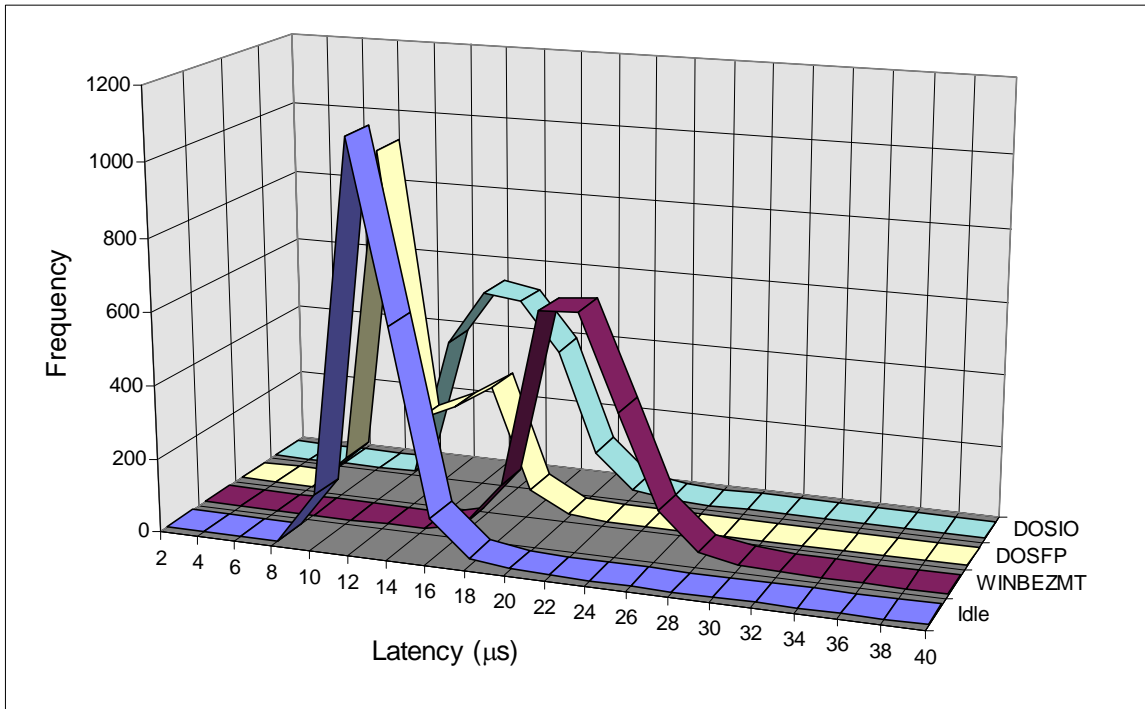


Figure 2 — Interrupt Latency Histogram

Note that for all four scenarios the interrupt latency is well localized.

What happens after the interrupt has been serviced?

VxD drivers for interrupt-driven devices are normally partitioned into an interrupt service routine and a segment of code that executes outside interrupt context in order to avoid executing too much code with interrupts disabled. For example, the interrupt service routine often operates on a small collection of buffers; when all of those buffers are exhausted, it schedules execution of its counterpart outside interrupt context. That code performs whatever actions are necessary to provide more buffers – often exchanging buffers with ring 3 code executing in the system VM – and the process repeats. Windows 95 offers three primary mechanisms for scheduling execution of code outside interrupt context: global events, VM events, and priority VM events. It is important to understand the circumstances that must hold before Windows 95 will execute code scheduled by each of these methods, and what latencies will potentially be encountered with each method.

Global events are the simplest method. Scheduling code execution with this method basically tells Windows 95 to execute the code in the context of any virtual machine once the system is out of interrupt context. You would expect this arrangement would lead to low latencies, which it does:

Test Case	Average Global Event Latency (µs)
Idle	43
WINBEZMT	64
DOSFP	29
DOSIO	66

Table 3

The distribution of these events can be relatively wide, however:

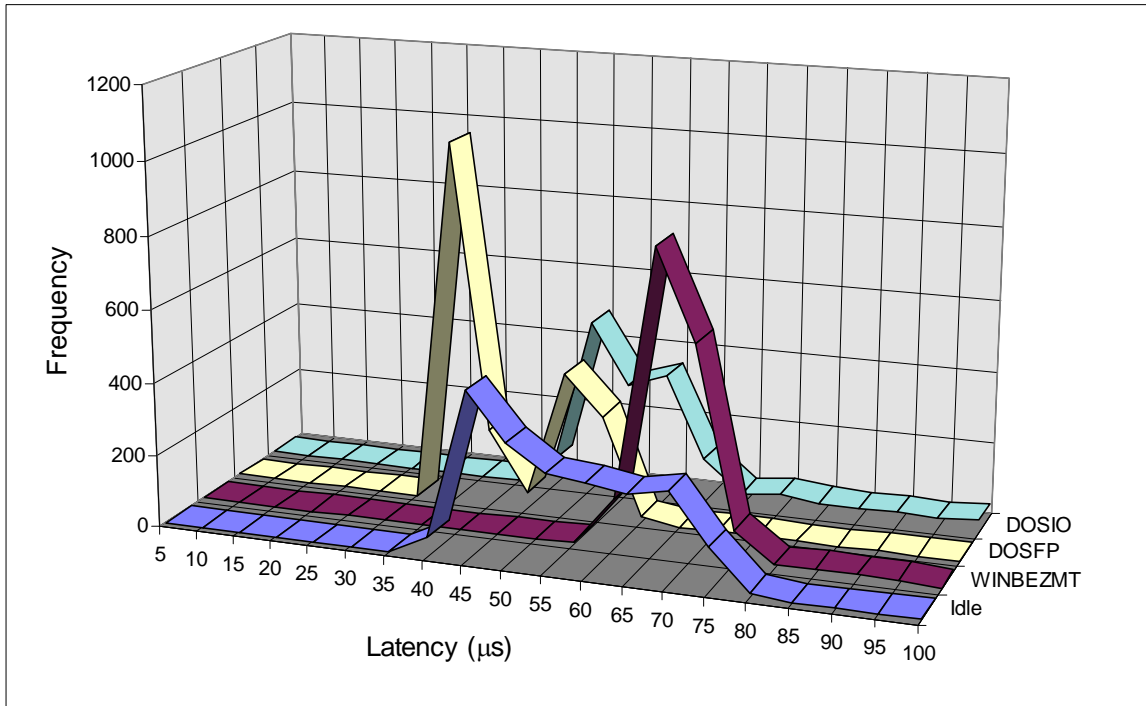


Figure 3 — Global Event Latency Histogram

Unfortunately, the fact that Windows 95 executes code scheduled via a global event in the context of any virtual machine limits its usefulness. VxD drivers are often paired with ring 3 driver code executing the system VM, and in this case global events are not of much use. If your non-interrupt context code can execute in any VM, however, the global event is the method of choice.

The next method for scheduling execution of code outside interrupt context is the VM event, which tells Windows 95 to execute the code within the context of a specified VM. If the current VM is not the specified target VM, there will be an additional latency contribution, as execution of the event code will be delayed until the target VM is the current VM. This is demonstrated in the latencies for the DOSFP and DOSIO test cases; the target VM for all test cases was the system VM.

Test Case	Average VM Event Latency (µs)
Idle	45
WINBEZMT	65
DOSFP	9291
DOSIO	9191

Table 4

This is also illustrated in Figure 4, where the period between bumps in the DOSFP and DOSIO curves is in the neighborhood of 10ms, which happens to be the minimum VM time slice granularity returned by the VMM's `Get_Time_Slice_Granularity` service. Each bump in the graph represents a state where the system VM was in a particular position within the queue of executable virtual machines when the event was scheduled; the first bump represents the scenario where the system VM was the next VM to be scheduled, the second bump represents the scenario where the system VM was the second VM in the queue, and so on. Note that the X-axis scale in Figure 4 has been adjusted to fit more of the DOSFP and DOSIO curves into the graph.

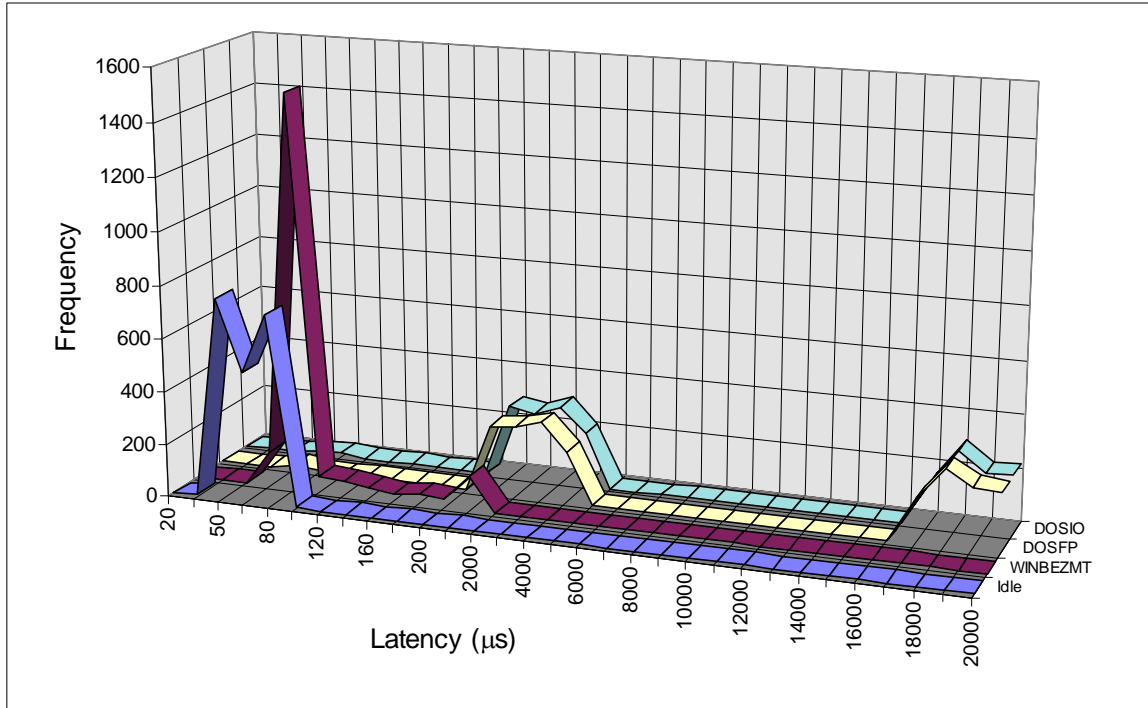


Figure 4 — VM Event Latency Histogram

The final method for scheduling execution of code outside interrupt context is the priority VM event, which is the same as a VM event except that the VMM boosts the priority of the specified VM in order to reduce the added latency:

Test Case	Average Priority VM Event Latency (µs)
Idle	60
WINBEZMT	100
DOSFP	192
DOSIO	280

Table 5

The corresponding histogram also reflects the reduced latency:

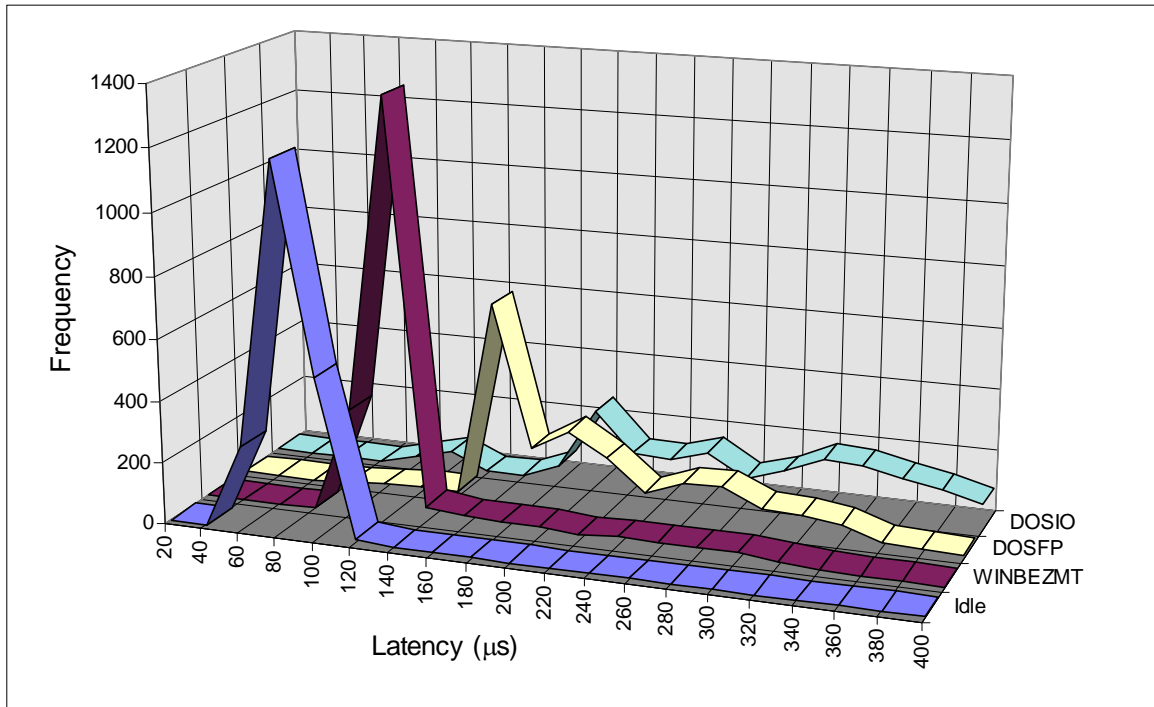


Figure 5 — Priority VM Event Latency Histogram

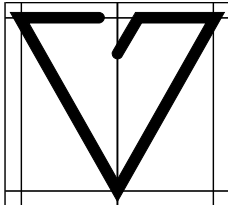
The priority VM event is a good choice for VxD drivers that have to service high-bandwidth hardware and also have to cooperate with code executing in ring 3 of the system VM.

Conclusions

Microsoft has made a concerted effort to make Windows 95 a better environment for hosting high-speed hardware that may have had trouble with long interrupt latencies under Windows 3.x. While still a long way from supporting the type of latencies expected of a real-time operating system, it is quite possible to support this type of hardware if careful design choices are made during driver development. There is still a significant risk, however, that a poorly-designed or otherwise uncooperative VxD device driver from another vendor could increase observed interrupt latency beyond acceptable limits for your hardware and its driver; unfortunately, under Windows 95 there is still no method for remedying this problem.

References

- King, Adrian. Inside Windows 95. Redmond, WA: Microsoft Press, 1994.
- Microsoft. STDVXD.DOC, part of the Windows 95 DDK. Redmond, WA: Microsoft, 1995.
- . VMM.DOC, part of the Windows 95 DDK. Redmond, WA: Microsoft, 1995.
- Pietrek, Matt. Windows Internals. Reading, MA: Addison-Wesley, 1993.
- Schulman, Andrew. Unauthorized Windows 95. Foster City, CA: IDG Books, 1994.
- Schulman, Andrew, David Maxey, and Matt Pietrek. Undocumented Windows. Reading, MA: Addison-Wesley, 1992.
- Thielen, David and Bryan Woodruff. Writing Windows Virtual Device Drivers. Reading, MA: Addison-Wesley, 1994.
- Vireo Software. VTOOLS User's Guide. Bolton, MA: Vireo Software, 1994.



Vireo Software

Vireo Software, Inc.

21 Half Moon Hill
Acton, MA 01720

Phone 508-264-9200
Fax 508-264-9205

Vireo@vireo.com
<http://world.std.com/~vireo>