Inside NT Interrupt Handling

Mark Russinovich

(Reprinted from WindowsItPro Magazine)

One of Windows NT's primary responsibilities is interfacing a computer to its peripheral devices. Like most modern operating systems, NT can dynamically integrate device driver programs to manage devices. Device drivers typically use interrupt signals to communicate with the devices they control. When a device has completed a driver-directed operation or when the device has new data for the driver, the device generates an interrupt signal. Depending on the state of the CPU, either a function within the driver immediately services the device's interrupt, or the CPU queues the interrupt for later servicing.

NT implements interrupt processing differently from many other operating systems, so how NT and NT device drivers process interrupts and how that processing affects other operations can be confusing. For example, systems programmers often ask me how interrupts affect thread scheduling--a natural question because the CPU can receive interrupt signals almost anytime, even while user programs perform ordinary processing. A common misconception is that NT won't service low-priority interrupt signals while high-priority threads are executing time-critical tasks.

The way NT handles interrupt processing affects NT's viability as an operating system for realtime (time-critical) environments (e.g., aircraft guidance systems). NT's rich development environment, user interface, and thread priority scheme make NT attractive to designers of realtime systems. However, realtime environments require the ability to predict how fast an operating system will react to interrupts; thus, the way NT implements interrupt handling affects how suitable it is for realtime applications.

In this column, I'll first provide background information about interrupts and describe NT's Interrupt Request Level (IRQL) architecture. Next, I'll present how device drivers register to receive notification of interrupts that their devices generate and what device drivers typically do upon notification. Finally, I'll describe the effect of interrupt processing on the NT scheduler and comment on how NT's interrupt processing affects its applicability for realtime systems.

Devices and Interrupts

All the major hardware architectures that NT runs on have interrupt-controller hardware to translate device interrupts into signal levels that feed into the CPU. The interrupt controller defines the interrupt priority scheme: When a device triggers an interrupt of a given priority, the controller masks (or withholds) from the CPU all interrupts of priority less than or equal to the device interrupt's priority. Until the CPU signals to the interrupt controller that it has finished servicing an interrupt, the interrupt controller pends (or puts on hold) lower priority interrupts but lets higher priority interrupts occur. When the interrupt level on the controller drops below an interrupt's priority, the controller lets the interrupt proceed to the CPU.

Device controllers connected to modern CPU buses (such as the popular PCI bus) dynamically determine which interrupts their device will use. When a device needs to inform its driver of an event (such as the availability of new data), it generates an interrupt that the device driver will recognize and acknowledge. The device driver registers an interrupt service routine (ISR) with the operating system, and the operating system executes the ISR in response to the interrupt. The ISR's job is to read status information from the device, acknowledge the interrupt, and stop the device from signaling the interrupt.

Interrupt Request Levels

NT manages interrupts by mapping interrupt-controller interrupt levels onto its hardware-independent table of interrupt levels. The hardware abstraction layer (HAL--the NT module custom-written for individual interrupt controllers, motherboards, or processor chip-sets) performs the mapping. In a

multiprocessor system, any processor can receive interrupts, so NT maintains an independent IRQL for each processor. A processor's IRQL represents the level of interrupt that the CPU is currently masking and directly corresponds to interrupts that the CPU's interrupt controller masks. Because NT's IRQLs are not tied to any hardware specification, NT can also map non-hardware interrupt types into its priority hierarchy. The operating system uses software interrupts primarily to initiate scheduling operations, such as thread switching, or I/O completion processing.

When NT services a hardware interrupt, NT sets the processor's IRQL to the corresponding mapped value in NT's IRQL table. NT programs the interrupt controller to mask out lower priority interrupts, and device drivers (as well as NT) can query the IRQL to determine its value. (NT permits some operations only when the IRQL is less than certain values, as we'll see later.)

The size of NT's IRQL table varies among processor architectures (Intel, Alpha, etc.) to better map the interrupt levels that standard interrupt controllers provide, but interrupt levels that device-driver designers and NT's developers might find interesting have symbolic names. Table 1 summarizes the symbolic IRQL names and their corresponding numeric values on Intel and Alpha architectures.

Table 1: IRQL Symbolic and Numeric Definitions

Symbolic Name	Purpose	Intel Level	Alpha Level
High Level	Highest interrupt level	31	7
Power Level	Power event	30	7
IPI Level	Interprocessor signal	29	6
Clock Level	Clock tick	28	5
Profile Level	Performance monitoring	27	3
Device Level	General device interrupts	3-26	3-4
Dispatch Level	Scheduler operations and deferred procedure calls (DPCs)	2	2
APC Level	Asynchronous procedure calls (APCs)	1	1
Passive Level	No interrupts	0	0

In Table 1, the lowest IRQL priority is Passive Level. When a processor is at this state, no interrupt processing activity is occurring. When code in user applications such as Word and Netscape is executing, the processor is at Passive Level. NT's goal is to return from higher IRQLs to Passive Level as quickly as possible so that NT can service new interrupts immediately and programs can get their work done.

The next two IRQLs above Passive Level (APC Level and Dispatch Level) are scheduler-related software interrupt levels. When the system is at APC Level, the executing thread will not receive asynchronous procedure call (APC) requests, which NT commonly uses for I/O cleanup operations. When the system decides that a scheduling decision needs to take place (e.g., when a thread's turn on the CPU ends), it issues a Dispatch Level software interrupt. (I'll describe the role of Dispatch Level software interrupts later in the article.)

All IRQLs higher than Dispatch Level relate to hardware interrupts. A system's peripheral devices (e.g., disk drives, keyboards, serial ports) have hardware interrupts mapped to IRQLs in the Device Level range. You can see in Table 1 that on Intel processors, the range is 3 through 26, and on Alpha

machines, the range is 3 through 4. The fact that such a difference exists between the two ranges implies that NT does not really prioritize general device interrupts. Even on Intel processors, where hardware interrupts might have different IRQL values, the assignments are arbitrary.

The IRQLs above Device Level have predefined associations with certain interrupts. Profile Level relates to the kernel profiling timer, Clock Level relates to the system clock tick, IPI Level relates to signals sent from one CPU to another, and Power Level relates to power failure events. NT reserves but does not currently use High Level.

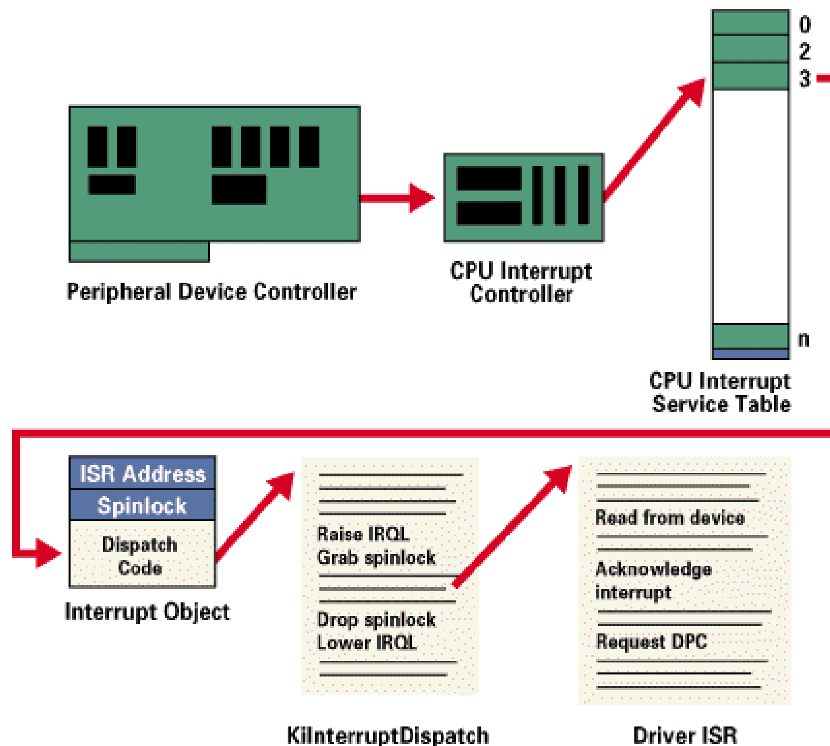
Interrupt Objects

Device drivers need a way to tell NT that they want specific functions executed when the processor receives interrupts associated with their devices. To satisfy this need, device drivers register an ISR with the I/O Manager by calling the IoConnectInterrupt subroutine. The parameters passed to IoConnectInterrupt describe all the attributes of the driver's ISR, including its address, the interrupt the ISR connects to, and whether other devices can share the same interrupt.

IoConnectInterrupt initializes an Interrupt Object to store information about the interrupt and its connected ISR. IoConnectInterrupt also programs the processor's interrupt hardware to point at code that IoConnectInterrupt places in the Interrupt Object. Thus, when the CPU receives the interrupt, control immediately transfers to the code in the Interrupt Object. This code calls NT's interrupt servicing helper function, KiInterruptDispatch, which raises the processor's IRQL, calls the appropriate ISR, and lowers the IRQL to its previous value. KiInterruptDispatch also obtains a spinlock specific to the interrupt and holds it while the ISR is running. A spinlock is a synchronization primitive commonly used in the NT kernel. The spinlock ensures that the ISR won't execute simultaneously on more than one processor (something that might cause device-driver writers some grief).

In NT, an ISR usually does nothing more than read a minimal amount of information from the interrupting device and acknowledge to the device that the driver has seen the interrupt. In other operating systems, ISRs often perform additional duties, such as fully processing an interrupt by reading large data buffers from or writing large data buffers to a device. However, one of NT's goals is to minimize time spent at high IRQLs, so NT postpones most interrupt servicing until the IRQL decreases. ISRs request a deferred procedure call (DPC) to inform the I/O Manager that they have work to do at a lower IRQL. A DPC is another function in the driver that the I/O Manager will call after the ISR finishes; the DPC performs most of the interaction with the driver's device.

Figure 1 depicts the typical flow of NT interrupt servicing. A device controller generates an interrupt signal on the processor bus that a processor interrupt controller handles. The signal causes the CPU to execute the code in the Interrupt Object registered for the interrupt; the code in turn calls the KiInterruptDispatch helper function. KiInterruptDispatch calls the driver's ISR, which requests a DPC.



NT also has a mechanism to handle interrupts not registered by device drivers. During system initialization, NT programs the interrupt controller to point at the controller's default ISRs. Default ISRs execute special processing when the system generates expected interrupts. For example, a page fault ISR must execute logic for situations in which programs reference virtual memory that does not have allocated space in the computer's physical memory. These situations might occur when programs interact with a file system to fetch data from a paging file or program image, or when programs reference an invalid address. NT programs unregistered interrupts to point at ISRs that recognize the system has generated an illegal interrupt. Most of these ISRs pop up a blue screen of death to inform a systems administrator that an illegal interrupt happened.

Deferred Procedure Calls

DPCs are the workhorses of NT's interrupt processing. NT tracks DPCs the same way it tracks interrupts--in objects. Device drivers usually initialize a DPC Object at the same time they connect to an interrupt. The information a driver must specify includes the address of its DPC function, the processor the DPC function needs to execute on, and the DPC's priority. By default, a DPC will always execute on the processor the ISR executes on; however, a device driver writer can override this assignment. In addition, DPCs default to medium priority (the choices are low, medium, and high), but a device driver writer can also control priority.

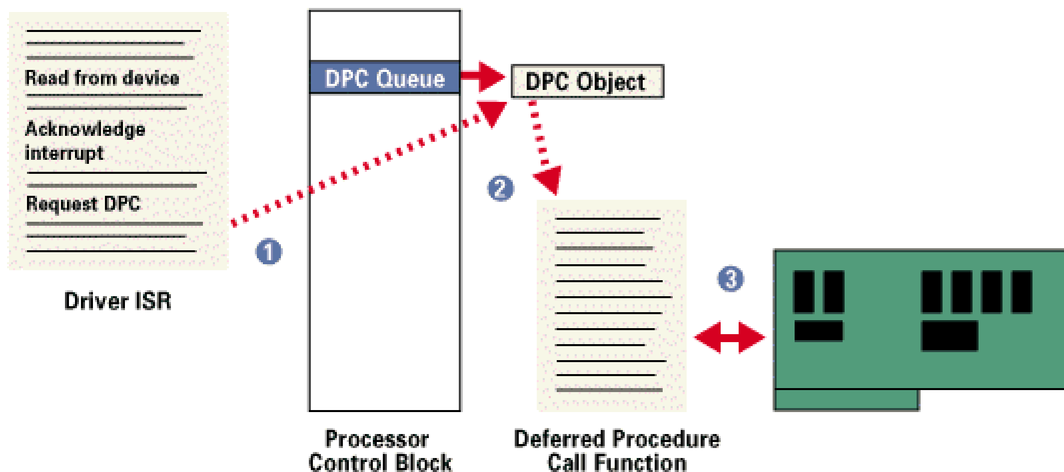
When an ISR requests a DPC, NT places the specified DPC Object on the target processor's DPC queue. If the DPC has low or medium priority, NT places the DPC Object at the end of the queue; if the DPC has high priority, NT inserts the DPC Object at the front of the queue. When the processor's IRQL is about to drop from Dispatch Level to a lower IRQL (APC Level or Passive Level), NT removes the DPC Objects from the DPC queue. NT ensures that the IRQL remains at Dispatch Level and pulls DPC Objects off the queue until the queue is empty (i.e., NT drains the queue), calling each DPC function in turn. Only when the queue is empty will NT let the IRQL drop below Dispatch Level and let regular thread execution continue.

DPC priorities can affect system behavior another way. NT usually initiates DPC queue draining with a software interrupt whose associated IRQL is Dispatch Level. NT generates such an interrupt only if the DPC is directed at the processor the ISR is requested on, and the DPC has high or medium priority. If the DPC has low priority, the DPC requests the interrupt only if the number of outstanding DPC requests for the processor rises above a threshold, or the number of DPCs requested on the processor within a time window is low. If a DPC is targeted at a CPU different from the one the ISR is running on and the DPC's priority is high, NT immediately signals the target CPU to drain its DPC queue. If the priority is medium or low, the number of DPCs queued on the target processor must exceed a threshold. The system idle thread also drains the DPC queue. Table 2, page 56, summarizes the situations initiating DPC queue draining.

Table 2: Situations Initiating DPC Queue Draining

DPC Priority	DPC Targeted at ISR's Processor	DPC Targeted at Another Processor
Low	DPC queue length exceeds maximum DPC queue length, DPC request rate is less than minimum DPC request rate, or system is idle	DPC queue length exceeds maximum DPC queue length or system is idle
Medium	Always	DPC queue length exceeds maximum DPC queue length or system is idle
High	Always	Always

Figure 2 shows a typical sequence of events. First, an ISR requests a DPC, and NT places the DPC Object on the DPC queue of the target processor. Depending on the DPC priority and the length of the DPC queue, NT generates a DPC software interrupt then or at some later time. When the processor drains the DPC queue, the DPC Object leaves the queue and control transfers to its DPC function, which completes interrupt processing by reading data from (or writing data to) the device that originated the interrupt.



Interrupts and Scheduling

The IRQL information in Table 1, page 56, shows that Dispatch Level is associated with scheduling operations. When the IRQL is at Dispatch Level or higher, NT masks scheduler software interrupts, which means that NT effectively turns off the scheduler. In fact, device drivers (and NT) must not

perform operations that require an immediate response by the scheduler when a processor is at an IRQL greater than or equal to Dispatch Level. This restriction includes doing anything that might indicate to NT that the current thread is giving up the CPU to wait for some event to occur because that action would cause the scheduler to find a new thread to execute. Another action that demands scheduler intervention is a page fault. When a thread accesses virtual memory that references data in the paging file, NT usually blocks the thread until the data is read. Therefore, at Dispatch Level or higher, NT does not permit access to memory not locked into the CPU's physical memory. If you've seen the IRQL_ NOT_LESS_OR_EQUAL blue screen stop code, you've probably witnessed the effect of a driver violating these rules.

Disabling the scheduler during interrupt processing has another, less-obvious effect: NT counts the time taken by ISRs and DPC functions against the quantum of the thread active at the time the CPU receives an interrupt. For instance, suppose Word is executing a spell-check operation, an interrupt comes in from a device, and the device's driver has a DPC that takes all the remaining time in Word's quantum (and then some). When the CPU's IRQL drops below Dispatch Level, the scheduler may decide to switch to a different thread in another application, effectively penalizing Word for the interrupt processing. Although this practice sounds unjust, most of the time NT distributes interrupt loads evenly across the applications in a system.

NT as a Realtime Operating System

Deadline requirements, either hard or soft, characterize realtime environments. Hard realtime systems (e.g., a nuclear power plant control system) have deadlines that the system must meet to avoid catastrophic failures such as loss of equipment or life. Soft realtime systems (e.g., a car's fuel-economy optimization system) have deadlines that the system can miss, but have timeliness as a desirable trait. In realtime systems, computers have sensor input devices and control output devices. The designer of a realtime computer system must know worst-case delays between the time an input device generates an interrupt and the time that the device's driver can control the output device to respond. This worst-case analysis must take into account the delays the operating system introduces, as well as the delays the application and device drivers impose.

Because NT does not prioritize device interrupts in any controllable way and user-level applications execute only when a processor's IRQL is Passive Level, NT is not always suitable as a realtime operating system. The system's devices and device drivers--not NT--ultimately determine the worst-case delay (the time from when an input device interrupts through when a realtime application processes the input and controls the output device). This factor becomes a problem when the designer of the realtime system uses off-the-shelf hardware. The designer can have difficulty determining how long every off-the-shelf device's ISR or DPC might take in the worst case. Even after testing, the designer cannot guarantee that a special case in a live system will not cause the system to miss an important deadline. Furthermore, the sum of all the delays a system's DPCs and ISRs can introduce usually far exceeds the tolerance of a time-sensitive environment.

Interrupt Management

Learning about how NT manages interrupts can help clear up confusion about how NT's interrupt processing affects thread scheduling and other operations: Thread priorities are essentially independent of interrupt priority levels. If you are a systems programmer, information about NT's interrupt management can help you understand how ISRs and DPCs fit into application processing. Finally, if you are considering NT as an operating system for a realtime environment, you can judge NT's suitability for your situation, or at least recognize some of the issues you need to consider before making a decision.