Scheduling, Thread Context, and IRQL

August 23, 2016

Abstract

This paper presents information about how thread scheduling, thread context, and a processor's current interrupt request level (IRQL) affect the operation of kernel-mode drivers for the Microsoft® Windows® family of operating systems. It is intended to provide driver writers with a greater understanding of the environment in which their code runs.

A companion paper, "Locks, Deadlocks, and Synchronization" at http://www.microsoft.com/whdc/hwdev/driver/LOCKS.mspx, builds on these fundamental concepts to address synchronization issues in drivers.

Contents

Introduction 3
Thread Scheduling 3
Thread Context and Driver Routines 4
Driver Threads 5
Interrupt Request Levels 6
  Processor-Specific and Thread-Specific IRQLs 8
    IRQL PASSIVE_LEVEL 8
    IRQL PASSIVE_LEVEL, in a critical region 9
    IRQL APC_LEVEL 9
    IRQL DISPATCH_LEVEL 10
    IRQL DIRQL 11
    IRQL HIGH_LEVEL 12
  Guidelines for Running at IRQL DISPATCH_LEVEL or Higher 13
  Changing the IRQL at which Driver Code Runs 13
  Standard Driver Routines, IRQL, and Thread Context 14
Interrupting a Thread: Examples 17
  Single-Processor Example 17
  Multiprocessor Example 18
Testing for IRQL Problems 20
  Techniques for Finding the Current IRQL 20
  PAGED_CODE Macro 20
  Driver Verifier Options 21
Best Practices for Drivers 21
Call to Action and Resources 21

Disclaimer

Introduction

Thread scheduling, thread context, and the current interrupt request level (IRQL) for each processor have important effects on how drivers work.

A thread's scheduling priority and the processor's current IRQL determine whether a running thread can be pre-empted or interrupted. In *thread pre-emption*, the operating system replaces the running thread with another thread, usually of higher thread priority, on the same processor. The effect of pre-emption on an individual thread is to make the processor unavailable for a while. In *thread interruption*, the operating system forces the current thread to temporarily run code at a higher interrupt level. The effect of interruption on an individual thread is similar to that of a forced procedure call.

Interruption and pre-emption both affect how code that runs in the thread can access data structures, use locks, and interact with other threads. Understanding the difference is crucial in writing kernel-mode drivers. To avoid related problems, driver writers should be familiar with:

- The thread scheduling mechanism of the operating system
- The thread context in which driver routines can be called
- The appropriate use of driver-dedicated and system worker threads
- The significance of various IRQLs and what driver code can and cannot do at each IRQL

Thread Scheduling

The Microsoft® Windows® operating system schedules individual threads, not entire processes, for execution. Every thread has a scheduling priority (its *thread priority*), which is a value from 0 to 31, inclusive. Higher numbers indicate higher priority threads.

Each thread is scheduled for a quantum, which defines the maximum amount of CPU time for which the thread can run before the kernel looks for other threads at the same priority to run. The exact duration of a quantum varies depending on what version of Windows is installed, the type of processor on which Windows is running, and the performance settings that have been established by a system administrator. (For more details, see *Inside Windows 2000*.)

After a thread is scheduled, it runs until one of the following occurs:

- Its quantum expires.
- It enters a wait state.
- A higher-priority thread becomes ready to run.

Kernel-mode threads do not have priority over user-mode threads. A kernel-mode thread can be pre-empted by a user-mode thread that has a higher scheduling priority.

Thread priorities in the range 1-15 are called *dynamic* priorities. Thread priorities in the range from 16-31 are called *real-time* priorities. Thread priority zero is reserved for the zero-page thread, which zeroes free pages for use by the memory manager.

Every thread has a *base priority* and a *current priority*. The base priority is usually inherited from the base priority for the thread's process. The current priority is the thread's priority at any given time. For kernel-mode driver code that runs in the context of a user thread, the base priority is the priority of the user process that originally requested the I/O operation. For kernel-mode driver code that runs in the context of a system worker thread, such as a work item, the base priority is the priority of the system worker threads that service its queue.

To improve system throughput, the operating system sometimes adjusts thread priorities. If a thread's base priority is in the d*ynamic range*, the operating system can temporarily increase ("boost") or decrease its priority, thus making its current priority different from its base priority. If a thread's base priority is in the *real-time range*, its current priority and base priority are always the same; threads running at real-time priorities never receive a priority boost. In addition, a thread that is running at a dynamic priority can never be boosted to a real-time priority. Therefore, applications that create threads with base priorities in the real-time range can be confident that these threads always have a higher priority than those in the dynamic range.

The system boosts a thread's priority when the thread completes an I/O request, when it stops waiting for an event or semaphore, or when it has not been run for some time despite being ready to run (called "CPU starvation"). Threads involved in the Graphical User Interface (GUI) and the user's foreground process also receive a priority boost in some situations. The amount of the increase depends on the reason for the boost and, for I/O operations, on the type of device involved. Drivers can affect the boost their code receives by:

- Specifying a priority boost in the call to **IoCompleteRequest**.
- Specifying a priority increment in the call to **KeSetEvent**, **KePulseEvent**, **KeReleaseSemaphore**.

Constants defined in ntddk.h and wdm.h indicate the appropriate priority boost for each device, event, and semaphore.

A thread's scheduling priority is not the same as the interrupt request level (IRQL) at which the processor operates.

Thread Context and Driver Routines

Most Windows drivers do not create threads; instead, a driver consists of a group of routines that are called in an existing thread that was created by an application or system component.

Kernel-mode software developers use the term "thread context" in two slightly different ways. In its narrowest meaning, thread context is the value of the thread's CONTEXT structure. The CONTEXT structure contains the values of the hardware registers, the stacks, and the thread's private storage areas. The exact contents and layout of this structure will vary according to the hardware platform. When Windows schedules a user thread, it loads information from the thread's CONTEXT structure into the user-mode address space.

From a driver developer's perspective, however, "thread context" has a broader meaning. For a driver, the thread context includes not only the values stored in the CONTEXT structure, but also the operating environment they define—particularly, the security rights of the calling application. For example, a driver routine might be called in the context of a user-mode application, but it can in turn call a **Zw*Xxx*** routine to perform an operation in the context of the operating system kernel. This paper uses "thread context" in this broader meaning.

The thread context in which driver routines are called depends on the type of device, on the driver's position in the device stack, and on the other activities currently in progress on the system. When a driver routine is called to perform an

I/O operation, the thread context might contain the user-mode address space and security rights of the process that requested the I/O. However, if the calling process was performing an operation on behalf of another user or application, the thread context might contain the user-mode address space and security rights of a different process. In other words, the user-mode address space might contain information that pertains to the process that requested the I/O, or it might instead contain information that pertains to a different process.

The dispatch routines of file system drivers (FSDs), file system (FS) filter drivers, and other highest-level drivers normally receive I/O requests in the context of the thread that initiated the request. These routines can access data in the user-mode address space of the requesting process, provided that they validate pointers and protect themselves against user-mode errors.

Most other routines in FSDs, FS filters, and highest-level drivers—and most routines in lower-level drivers—are called in an arbitrary thread context. Although the highest-level drivers receive I/O requests in the context of the requesting thread, they often forward those requests to their lower level drivers on different threads. Consequently, you can make no assumptions about the contents of the user-mode address space at the time such routines are called.

For example, when a user-mode application requests a synchronous I/O operation, the highest-level driver's I/O dispatch routine is called in the context of the thread that requested the operation. The dispatch routine queues the I/O request for processing by lower-level drivers. The requesting thread then enters a wait state until the I/O is complete. A different thread de-queues the request, which is handled by lower-level drivers that run in the context of whatever thread happens to be executing at the time they are called.

A few driver routines run in the context of a system thread. System threads have the address space of the system process and the security rights of the operating system itself. Work items queued with the **Io*Xxx*WorkItem** routines run in a system thread context, and so do all **DriverEntry** and *AddDevice* routines. No user-mode requests arrive in a system thread context.

The section "Standard Driver Routines, IRQL, and Thread Context," later in this paper, lists the thread context in which each standard driver routine is called.

Driver Threads

Although a driver can create a new, driver-dedicated thread by calling **PsCreateSystemThread**, drivers rarely do so. Switching thread context is a relatively time-consuming operation that can degrade driver performance if it occurs often. Therefore, drivers should create dedicated threads only to perform continually repeated or long-term activities, such as polling a device or managing multiple data streams, as a network driver might do.

To perform a short-term, finite task, a driver should not create its own thread; instead, it can temporarily "borrow" a system thread by queuing a work item. The system maintains a pool of dedicated threads that all drivers share. When a driver queues a work item, the system dispatches it to one of these threads for execution. Drivers use work items to run code in the kernel address space and security context or to call functions that are available only at IRQL PASSIVE_LEVEL. For example, a driver's *IoCompletion* routine (which can run at IRQL DISPATCH_LEVEL) should use a work item to call a routine that runs at IRQL PASSIVE_LEVEL.

To queue a work item, a driver allocates an object of type IO_WORKITEM and calls the **IoQueueWorkItem** routine, specifying the callback routine to perform the task and the queue in which to place the work item. The kernel maintains three queues for work items:

- Delayed work queue. Items in this queue are processed by a system worker thread that has a variable, dynamic thread priority. Drivers should use this queue.
- Critical work queue. Items in this queue are processed by a system worker thread at a higher thread priority than the items in the delayed work queue.
- Hypercritical work queue. Items in this queue are processed by a system worker thread at a higher priority than items in the critical work queue. This work queue is reserved for use by the operating system and must not be used by drivers.

A system worker thread removes the work item from the queue and runs the driver-specified callback routine in a system thread context at IRQL PASSIVE_LEVEL. The operating system ensures that the driver is not unloaded while the callback routine is running. To synchronize the actions of the callback routine with other driver routines, the driver can use one of the Windows synchronization mechanisms. For more information about synchronization, see the companion white paper, “Locks, Deadlocks, and Synchronization.”

Because the system has a limited supply of dedicated worker threads, the tasks assigned to them should be completed quickly. For example, a driver should not have a work item that runs continuously until the driver is unloaded. Instead, the driver should queue a work item only when it is needed, and the work item routine should exit when it has completed its work. For the same reasons, drivers should never include an infinite loop (such as might occur in a file system driver) in a work item. Drivers should also avoid queuing excessive numbers of work items, because tying up the system worker threads can deadlock the system. Instead of queuing a separate work item routine for each individual operation, the driver should have a single work item routine that performs any outstanding work and then exits when there is no more immediate work to perform.

Interrupt Request Levels

An *interrupt request level (IRQL)* defines the hardware priority at which a processor operates at any given time. In the Windows Driver Model, a thread running at a low IRQL can be interrupted to run code at a higher IRQL.

The number of IRQLs and their specific values are processor-dependent. The IA64 and AMD64 architectures have 16 IRQLs and the x86-based processors have 32. (The difference is due primarily to the types of interrupt controllers that are used with each architecture.) Table 1 is a list of the IRQLs for x86, IA64, and AMD64 processors.

Table 1. Interrupt Request Levels

IRQL	**IRQL value**			**Description**
	x86	IA64	AMD64	
PASSIVE_LEVEL	0	0	0	User threads and most kernel-mode operations
APC_LEVEL	1	1	1	Asynchronous procedure calls and page faults
DISPATCH_LEVEL	2	2	2	Thread scheduler and deferred procedure calls (DPCs)
CMC_LEVEL	N/A	3	N/A	Correctable machine-check level (IA64 platforms only)

IRQL	IRQL value			Description
Device interrupt levels (DIRQL)	3-26	4-11	3-11	Device interrupts
PC_LEVEL	N/A	12	N/A	Performance counter (IA64 platforms only)
PROFILE_LEVEL	27	15	15	Profiling timer for releases earlier than Windows 2000
SYNCH_LEVEL	27	13	13	Synchronization of code and instruction streams across processors
CLOCK_LEVEL	N/A	13	13	Clock timer
CLOCK2_LEVEL	28	N/A	N/A	Clock timer for x86 hardware
IPI_LEVEL	29	14	14	Interprocessor interrupt for enforcing cache consistency
POWER_LEVEL	30	15	14	Power failure
HIGH_LEVEL	31	15	15	Machine checks and catastrophic errors; profiling timer for Windows XP and later releases

When a processor is running at a given IRQL, interrupts at that IRQL and lower are masked off (blocked) on the processor. For example, a processor that is running at IRQL=DISPATCH_LEVEL can be interrupted only by a request at an IRQL greater than DISPATCH_LEVEL.

The system schedules all threads to run at IRQLs below DISPATCH_LEVEL, and the system's thread scheduler itself (also called "the dispatcher") runs at IRQL=DISPATCH_LEVEL. Consequently, a thread that is running at or above DISPATCH_LEVEL has, in effect, exclusive use of the current processor. Because DISPATCH_LEVEL interrupts are masked off on the processor, the thread scheduler cannot run on that processor and thus cannot schedule any other thread.

On a multiprocessor system, each processor can be running at a different IRQL. Therefore, one processor could run a driver's *InterruptService* routine at DIRQL while a second processor runs driver code in a worker thread at PASSIVE_LEVEL. Because more than one thread could thus attempt to access shared data simultaneously, drivers must protect shared data by using an appropriate synchronization method. Drivers should use a lock that raises the IRQL to the highest level at which any code that accesses the data can run. For example, a driver uses a spin lock to protect data that can be accessed at IRQL=DISPATCH_LEVEL. For more information about synchronization mechanisms, see the companion white paper, "Locks, Deadlocks, and Synchronization."

On a single-processor system, raising the IRQL to DISPATCH_LEVEL or higher has the same effect as using a spin lock because raising the IRQL can prevent the pre-emption or interruption of the currently executing code. For example, when a driver's *StartIo* routine is running at DISPATCH_LEVEL on a single-processor system, other driver code that runs at APC_LEVEL or PASSIVE_LEVEL cannot run until the IRQL drops. Similarly, when a driver's *InterruptService* routine is running at DIRQL, the DPC queued by that routine cannot run until the *InterruptService* routine exits. In fact, the operating system's spin lock acquisition and release routines raise the IRQL on single-processor systems; they do not actually manipulate a lock object. On multiprocessor machines, however, spin lock acquisition routines raise the IRQL on the current processor while other processors spin on the lock.

For detailed information about locking, see the companion white paper, “Locks, Deadlocks, and Synchronization.”

Processor-specific and Thread-specific IRQLs

As previously mentioned, the system’s thread scheduler runs at IRQL=DISPATCH_LEVEL. IRQLs at or above DISPATCH_LEVEL are processor specific. Hardware and software interrupts at these levels are targeted at individual processors. The following processor-specific IRQLs are commonly used by drivers:

- DISPATCH_LEVEL
- DIRQL
- HIGHEST_LEVEL

IRQLs below DISPATCH_LEVEL are thread specific. Software interrupts at these levels are targeted at individual threads. Drivers use the following thread-specific IRQLs:

- PASSIVE_LEVEL
- APC_LEVEL

The thread scheduler considers only thread priority, and not IRQL, when preempting a thread. If a thread running at IRQL=APC_LEVEL blocks, the scheduler might select a new thread for the processor that was previously running at PASSIVE_LEVEL.

Although only two thread-specific IRQL values are defined, the system actually implements three levels. The system implements an intermediate level between PASSIVE_LEVEL and APC_LEVEL. Code running at this level is said to be in a *critical region*. Code that is running at IRQL=PASSIVE_LEVEL calls **KeEnterCriticalRegion** to raise the IRQL to this level and calls **KeLeaveCriticalRegion** to return the IRQL to PASSIVE_LEVEL.

The following sections provide more information about the operating environment for driver code at each of these levels.

IRQL PASSIVE_LEVEL

While the processor is operating at PASSIVE_LEVEL, the operating system uses the scheduling priorities of the current threads to determine which thread to run. PASSIVE_LEVEL is the processor’s normal operating state, at which any interrupt can occur. Any thread that is running at PASSIVE_LEVEL is considered pre-emptible because it can be replaced by a thread that has a higher scheduling priority. A thread that is running at PASSIVE_LEVEL is also considered interruptible because it can be interrupted by a request at a higher IRQL.

Occasionally, driver code that is running at IRQL PASSIVE_LEVEL must call a system service routine or perform some other action that requires running at a higher IRQL (usually DISPATCH_LEVEL). Before making the call or performing the action, the driver must raise its IRQL to the required level; immediately after completing the action, the driver must lower the IRQL.

Code that is running at PASSIVE_LEVEL is considered to be working on behalf of the current thread. An application that creates a thread can suspend that thread while the thread is running kernel-mode code at PASSIVE_LEVEL. Therefore, driver code that acquires a lock at IRQL=PASSIVE_LEVEL must ensure that the thread in which it is running cannot be suspended while it holds the lock; thread suspension would disable access the to driver's device. This problem is usually resolved by using locks that raise the IRQL. Another solution is to enter a critical region whenever it tries to acquire such a lock. This issue is covered in greater detail in the companion paper, "Locks, Deadlocks, and Synchronization."

IRQL PASSIVE_LEVEL in a critical region

Code that is running at PASSIVE_LEVEL in a critical region is effectively running at an intermediate level between PASSIVE_LEVEL and APC_LEVEL. Calls to **KeGetCurrentIrql** return PASSIVE_LEVEL. Driver code can determine whether it is operating in a critical region by calling the function **KeAreApcsDisabled** (available in Windows XP and later releases).

Driver code that is running above PASSIVE_LEVEL (either at PASSIVE_LEVEL in a critical region or at APC_LEVEL or higher) cannot be suspended. Almost every operation that a driver can perform at PASSIVE_LEVEL can also be performed in a critical region. Two notable exceptions are raising hard errors and opening a file on storage media.

IRQL APC_LEVEL

APC_LEVEL is a thread-specific IRQL that is most commonly associated with paging I/O. Applications cannot suspend code that is running at IRQL=APC_LEVEL. The system implements fast mutexes (a type of synchronization mechanism) at APC_LEVEL. The **KeAcquireFastMutex** routine raises the IRQL to APC_LEVEL, and **KeReleaseFastMutex** returns the IRQL to its original value.

The only difference between a thread that is running at PASSIVE_LEVEL with APCs disabled and a thread that is running at APC_LEVEL is that while running at APC_LEVEL, the thread cannot be interrupted to deliver a special kernel-mode APC.

Asynchronous Procedure Calls (APCs)

Asynchronous procedure calls (APCs) are software interrupts that are targeted at a specific thread. The system uses APCs to perform work in the context of a particular thread, such as writing back the status of an I/O operation to the requesting application.

How a target thread responds to APCs depends on the thread's state and the type of APC. The following briefly summarizes the actions; for a complete description, see "Do Waiting Threads Receive Alerts and APCs" under "Synchronization" in the "Kernel-Mode Drivers Architecture Design Guide" of the Windows Driver Development Kit (DDK).

Every thread has two kernel-mode APC queues, one for APC_LEVEL callbacks and another for critical region callbacks. Each time the system adds an APC to a queue, it checks to see whether the target thread is currently running. If so, the system requests an interrupt on the appropriate processor. If the thread is still running when the system services the interrupt, the APC runs immediately, if appropriate. If the target thread is not running, the APC is added to the queue and runs the next time the thread is scheduled; the interrupt does not cause the target thread to run immediately. If the current IRQL is too high to run the APC, the APC runs the next time the IRQL is lowered below the level of the APC. If the thread is waiting at a lower IRQL, the system wakes the thread temporarily to deliver the APC, and then the thread resumes waiting.

Both user-mode code and kernel-mode code can issue APCs. The system defines three types of APCs:

- User-mode APCs
- Normal kernel-mode APCs
- Special kernel-mode APCs

User-mode APCs are primarily used in completing I/O operations. Win32® APIs such as **ReadFileEx** and **WriteFileEx** allow the caller to specify an I/O completion callback routine. To run the callback routine, the system queues a user-mode APC to the thread that requested the I/O. A user-mode application can queue a user-mode APC directly by calling the Win32 API **QueueUserAPC**. User-mode APCs are beyond the scope of this document.

Normal kernel-mode APCs are delivered at the intermediate level that corresponds to PASSIVE_LEVEL in a critical region. The system delivers a normal kernel-mode APC when the target thread is already running at PASSIVE_LEVEL or when the thread is returning to PASSIVE_LEVEL after exiting from a critical region or after lowering the IRQL. Normal kernel-mode APCs have two routines, a *Special Routine* that runs at APC_LEVEL, and a *Normal Routine* that subsequently runs at PASSIVE_LEVEL in a critical region. The special routine typically frees the APC structure.

Special kernel-mode APCs are delivered at APC_LEVEL. The system delivers them if the target thread is running at an IRQL below APC_LEVEL or if the target thread is returning to an IRQL below APC_LEVEL.

Normal kernel-mode APCs and special kernel-mode APCs are queued when kernel-mode operating system code calls an undocumented internal routine; drivers cannot queue kernel-mode APCs directly.

The I/O manager queues the special kernel-mode APC for I/O completion whenever an I/O request completes. When a device's drivers complete a buffered I/O request, the I/O Manager queues this APC to the user-mode thread that originated the I/O request. When the APC runs, the operating system restores the thread's context and the I/O manager copies data from the driver's kernel-space output buffer to the requesting thread's user-space buffer.

IRQL DISPATCH_LEVEL

DISPATCH_LEVEL is the highest software interrupt level and the first processor-specific level. The Windows thread scheduling and dispatching components (collectively called "the dispatcher") run at IRQL DISPATCH_LEVEL. Some other kernel-mode support routines, some driver routines, and all deferred procedure calls (DPCs) also run at IRQL DISPATCH_LEVEL. While the processor operates at this level, one thread cannot pre-empt another; only a hardware interrupt can interrupt the running thread. To maximize overall system throughput, driver code that runs at DISPATCH_LEVEL should perform only the minimum amount of required processing.

Because code that is running at DISPATCH_LEVEL cannot be pre-empted, the operations that a driver can perform at DISPATCH_LEVEL are restricted. Any code that must wait for an object that another thread sets or signals asynchronously—such as an event, semaphore, mutex, or timer—cannot run at DISPATCH_LEVEL because the waiting thread cannot block while waiting for the other thread to perform the action. Waiting for a nonzero period on such an object while at DISPATCH_LEVEL causes the system to deadlock and eventually to crash.

Deferred procedure calls (DPCs) are, in effect, software interrupts targeted at processors. DPCs (including *DpcForIsr*, *CustomDpc*, and *CustomTimerDpc* routines) are always called at IRQL DISPATCH_LEVEL in an arbitrary thread context. Drivers usually use DPCs for the following:

- To perform additional processing after a device interrupts. Such DPCs are either *DpcForIsr* or *CustomDpc* routines that are queued by the driver's *InterruptService* routine.
- To handle device time-outs. Such a DPC is a *CustomTimerDpc* routine that is queued when the timer expires by the **KeSetTimer** or **KeSetTimerEx** routine.

The kernel maintains a queue of DPCs for each processor and runs DPCs from this queue just before the processor's IRQL drops below DISPATCH_LEVEL.

Each DPC is assigned to the queue for the same processor on which the code that queues it is running. The kernel removes DPC objects from the head of the queue and adds them to its tail. A driver can change the processor for which a DPC object is queued by calling **KeSetTargetProcessorDpc**. A driver can also change the DPC's relative location in the queue by calling **KeSetImportanceDpc**. However, drivers rarely need to change either of these characteristics.

If a device interrupts while either its *DpcForIsr* or *CustomDpc* routine is running, its *InterruptService* routine pre-empts the DPC and queues a DPC object as it normally would. In a single-processor system, the DPC object is placed at the end of the single DPC queue, where it runs in sequence with any other DPCs in the queue after the *InterruptService* routine and the current DPC complete. In a multi-processor system, however, the second interrupt could occur on a different processor.

For example, assume a device interrupts on Processor 1 while its *DpcForIsr* routine is running on Processor 0. The system runs the *InterruptService* routine on Processor 1 to handle the interrupt. When the *InterruptService* routine queues its *DpcForIsr* routine, the system places the DPC object into the DPC queue of Processor 1. Thus, a driver's *InterruptService* routine can run at the same time as its DPC routine, and the same DPC routine can run on two or more processors at the same time. If both routines attempt to access the same data simultaneously, serious errors can occur. Drivers must use spin locks to protect shared data in these scenarios.

IRQL DIRQL

DIRQL describes the range of IRQLs that physical devices can generate. Each processor architecture has a range of DIRQLs, as shown in Table 1, "Interrupt Request Levels." The DIRQL for each device instance is available to its driver in the CM_RESOURCE_LIST structure passed by the PnP manager as part of the IRP_MN_START_DEVICE request. The driver, in turn, passes this IRQL to the I/O manager when it calls **IoConnectInterrupt** to connect its interrupt object. Multiple devices can interrupt at the same DIRQL.

Note

Microsoft has made several enhancements to the interrupt architecture in the next release of Windows, codenamed "Longhorn." For information about these pending changes, see the white paper "Interrupt Architecture Enhancements in Microsoft Windows, Codenamed Longhorn," which is available at http://www.microsoft.com/whdc/hwdev/bus/pci/MSI.mspx.

Two types of driver routines run at DIRQL:

- *InterruptService* routines
- *SynchCritSection* routines

Function drivers for physical devices usually include these routines; filter and file system drivers never do.

InterruptService routines must run at DIRQL so that they can handle the current interrupt without receiving further interrupts from the interrupt controller. A driver's *InterruptService* routine should first determine whether its device is the source of the interrupt. If so, the routine stops the device from generating further interrupts, saves any necessary context information, and queues a deferred procedure call (DPC) to run later at DISPATCH_LEVEL. If the interrupt was generated by some other device, the routine should simply return FALSE.

A driver's *InterruptService* routine runs on the same processor on which its device interrupted; in turn, its *DpcForIsr* (or *CustomDpc*) routine runs on the same processor as the *InterruptService* routine that queued it.

InterruptService routines must follow these important rules:

- The *InterruptService* routine must not return FALSE if its device generated the interrupt. Such "unclaimed" interrupts can eventually hang or crash the system.
- The *InterruptService* routine must not access the device hardware when the device is in a low power state that does not support such access. To prevent such problems, drivers should disconnect interrupts when transitioning their devices out of the D0 power state.

InterruptService routines should perform only the tasks that cannot be deferred until the processor is at a lower IRQL. *InterruptService* routines that run for longer than a minimal time can reduce performance across the operating system.

SynchCritSection routines also run at DIRQL. A driver uses a *SynchCritSection* routine to access data that is shared with an *InterruptService* routine. Like *InterruptService* routines, *SynchCritSection* routines should perform only the minimum set of required tasks: accessing hardware registers, writing data that is shared with the *InterruptService* routine, and so forth. For example, a driver might need a *SynchCritSection* routine to re-enable device interrupts from its *DpcForIsr* routine.

A driver cannot call a *SynchCritSection* routine directly. Instead, the driver calls **KeSynchronizeExecution**, passing a pointer to the *SynchCritSection* routine. **KeSynchronizeExecution** raises the IRQL on the processor to DIRQL for the device, acquires the device's interrupt spin lock, and then starts the routine. Before returning to the caller, **KeSynchronizeExecution** releases the interrupt spin lock and lowers the IRQL on the current processor to its previous value.

While running at DIRQL, driver code must conform to the guidelines described in the section "Guidelines for Running at IRQL DISPATCH_LEVEL or Higher."

IRQL HIGH_LEVEL

Certain bug-check and non-maskable interrupt (NMI) callback routines run at IRQL HIGH_LEVEL. Because no interrupts can occur at IRQL HIGH_LEVEL, these routines are guaranteed to run without interruption.

The lack of interrupts, however, means that actions of the callback routines are severely restricted. In addition to the restrictions listed in the section "Guidelines for Running at IRQL DISPATCH_LEVEL or Higher," the following rules apply to code that runs at HIGH_LEVEL:

- The code must not allocate memory.
- The code must not use any synchronization mechanisms.
- The code must not call any routines that run at IRQL<= DISPATCH_LEVEL.

Guidelines for Running at IRQL DISPATCH_LEVEL or Higher

Driver code that runs at IRQL DISPATCH_LEVEL or above must conform to the following guidelines:

- Use only nonpageable data and code; do not perform any actions that require paging. The operating system must wait for paging I/O operations to complete, and such waits cannot be performed at DISPATCH_LEVEL or higher. (For the same reason, any driver routine that obtains a spin lock must not be pageable.) A driver can store data that it will access at IRQL>=DISPATCH_LEVEL in the following locations:
    - The device object, usually in the device extension.
    - The kernel stack, for small amounts of data that do not need to persist beyond the lifetime of the function.
    - Nonpaged memory allocated by the driver. For large amounts of data, such as that required for I/O buffers, drivers should use the **ExAllocate*Xxx*** or **MmAllocate*Xxx*** routines, as appropriate.
- Never wait for a nonzero period on a kernel dispatcher object (an event, semaphore, timer, kernel mutex, thread, process, or file object).
- Do not call routines that convert strings from ANSI to UNICODE, or vice versa. These routines are in pageable code. Furthermore, most of the **Rtl*Xxx*String** routines can be called only from PASSIVE_LEVEL. Check the Windows DDK documentation before calling such routines at or above DISPATCH_LEVEL.
- Never call **KeReleaseSpinLock** unless you have previously called **KeAcquireSpinLock**. Similarly, never call **KeReleaseSpinLockFromDpcLevel** unless you have previously called **KeAcquireSpinLockAtDpcLevel**. You cannot mix these two types of spin lock calls.
- Never call **KeAcquireSpinLock** from code that is running at IRQL = DISPATCH_LEVEL because **KeAcquireSpinLock** raises the current IRQL to DISPATCH_LEVEL. Instead, use **KeAcquireSpinLockAtDpcLevel**, which does not change the current IRQL.

Changing the IRQL at which Driver Code Runs

In general, the IRQL at which the operating system calls a driver routine is appropriate for the tasks that such a routine must perform. For example, a *DpcForIsr* routine usually must call **IoStartNextPacket**, which in turn calls the driver's *StartIo* routine. The *DpcForIsr*, **IoStartNextPacket**, and *StartIo* routines must all be called at DISPATCH_LEVEL to ensure that the I/O operations they perform complete without pre-emption by other user threads.

In some situations, however, driver code must raise the IRQL so that the driver can call a routine at DISPATCH_LEVEL. For example, if a driver calls **KeGetCurrentProcessorNumber** at IRQL< DISPATCH_LEVEL, a processor switch could occur between instructions, thus returning the incorrect value to the caller. Therefore, drivers sometimes must call **KeRaiseIrql** before calling **KeGetCurrentProcessorNumber**.

Although a driver can safely raise the IRQL when necessary, a driver must NEVER lower the IRQL without first raising it. Furthermore, a driver routine must never lower the IRQL below the setting at which it was called. Doing so can disrupt operations that the caller of the driver routine was relying on to complete atomically; such a disruption usually causes the system to crash.

Occasionally, however, driver code that runs at IRQL>=DISPATCH_LEVEL must communicate with code at a lower IRQL. For example, a driver might need to issue a synchronous device control request to its device after completion of an I/O operation. *IoCompletion* routines can be called at IRQL = DISPATCH_LEVEL, but **IoBuildDeviceIoControlRequest** must be called at PASSIVE_LEVEL. In this situation, the driver should use the **IoAllocateWorkItem** and **IoQueueWorkItem** routines (which can be called at IRQL = DISPATCH_LEVEL) to allocate and queue a work item routine that builds and sends the device control request. The work item routine will be called in the context of a system thread at IRQL = PASSIVE_LEVEL.

The following are guidelines for changing IRQL:

- Never call **KeRaiseIrql** with an IRQL that is lower than the current IRQL.
- Never call **KeLowerIrql** unless you have previously called **KeRaiseIrql**. The two calls must occur within the same function. A function must always return at the same IRQL at which it was called.
- Never call **KeLowerIrql** with an IRQL lower than the IRQL at which you called **KeRaiseIrql**.

Standard Driver Routines, IRQL, and Thread Context

Table 2 is a list of the standard driver routines, the IRQL at which each routine is called, and the thread context in which the routine runs. In addition to the routines that are listed here, there are many device-type-specific driver routines that are called at DISPATCH_LEVEL. For details, see the device-specific documentation in the Windows DDK.

Table 2. IRQL and Thread Context for Standard Driver Routines

Routine	Caller's IRQL	Thread context
AdapterControl	DISPATCH_LEVEL	Arbitrary
AdapterListControl	DISPATCH_LEVEL	Arbitrary
AddDevice	PASSIVE_LEVEL	System
BugCheckCallback	HIGH_LEVEL	Arbitrary: depends on state of operating system when the bug check occurred
BugCheckDumpIoCallback	HIGH_LEVEL	Arbitrary: depends on state of operating system when the bug check occurred
BugCheckSecondaryDumpDataCallback	HIGH_LEVEL	Arbitrary: depends on state of operating system when the bug check occurred
Cancel	DISPATCH_LEVEL	Arbitrary
ControllerControl	DISPATCH_LEVEL	Arbitrary
CsqAcquireLock	IRQL of the routine that called IoCsqXxx. Usually <= DISPATCH_LEVEL	Arbitrary
CsqCompleteCanceledIrp	<= DISPATCH_LEVEL	Arbitrary

Routine	Caller's IRQL	Thread context
CsqInsertIrp	IRQL of the lock acquired by CsqAcquireLock. Usually <= DISPATCH_LEVEL	Arbitrary
CsqInsertIrpEx	IRQL of the lock acquired by CsqAcquireLock. Usually <= DISPATCH_LEVEL	Arbitrary
CsqPeekNextIrp	IRQL of the lock acquired by CsqAcquireLock. Usually <= DISPATCH_LEVEL	Arbitrary
CsqReleaseLock	IRQL of the lock acquired by CsqAcquireLock. Usually <= DISPATCH_LEVEL	Arbitrary
CsqRemoveIrp	IRQL of the lock acquired by CsqAcquireLock. Usually <= DISPATCH_LEVEL	Arbitrary
CustomDpc	DISPATCH_LEVEL	Arbitrary
CustomTimerDpc	DISPATCH_LEVEL	Arbitrary
DispatchCleanup	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchClose (for FSD, FS filters, and other highest-level drivers)	APC_LEVEL	Arbitrary
DispatchClose (for all other drivers)	PASSIVE_LEVEL	Arbitrary
DispatchCreate	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchCreateClose	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchDeviceControl (for devices not in paging path)	PASSIVE_LEVEL	Non-arbitrary for FSD and FS filters; arbitrary for other drivers
DispatchDeviceControl (for devices in paging path)	<= DISPATCH_LEVEL	Arbitrary
DispatchFlushBuffers	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers

Routine	Caller's IRQL	Thread context
DispatchInternalDeviceControl	Depends on the device type, but always <= DISPATCH_LEVEL	Arbitrary
DispatchPnp	PASSIVE_LEVEL	Arbitrary
DispatchPower (if the DO_POWER_PAGABLE flag is not set in the device object)	<= DISPATCH_LEVEL	Arbitrary
DispatchPower (if the DO_POWER_PAGABLE flag is set in the device object)	PASSIVE_LEVEL	Arbitrary
DispatchQueryInformation	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchRead	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchRead (for devices in paging path)	APC_LEVEL	Arbitrary
DispatchRead and *DispatchWrite* routines of drivers in the storage stack	<= DISPATCH_LEVEL	Arbitrary
DispatchReadWrite (for devices not in paging path)	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchReadWrite (for devices in paging path)	APC_LEVEL	Arbitrary
DispatchSetInformation	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchShutdown	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DispatchSystemControl	PASSIVE_LEVEL	Arbitrary
DispatchWrite (for devices in paging path)	APC_LEVEL	Arbitrary
DispatchWrite (for devices not in paging path)	PASSIVE_LEVEL	Non-arbitrary for FSD, FS filter, and other highest-level drivers; arbitrary for other drivers
DllInitialize	PASSIVE_LEVEL	System or arbitrary
DllUnload	PASSIVE_LEVEL	Arbitrary
DpcForIsr	DISPATCH_LEVEL	Arbitrary

Routine	Caller's IRQL	Thread context
DriverEntry	PASSIVE_LEVEL	System
InterruptService	DIRQL for the associated interrupt object	Arbitrary
IoCompletion	<= DISPATCH_LEVEL	Arbitrary
IoTimer	DISPATCH_LEVEL	Arbitrary
Reinitialize	PASSIVE_LEVEL	System
StartIo	DISPATCH_LEVEL	Arbitrary
SynchCritSection	DIRQL for the associated interrupt object	Arbitrary
Unload	PASSIVE_LEVEL	System

Interrupting a Thread: Examples

Some simple examples can show what happens when a thread is pre-empted or interrupted. This section presents a single-processor example and a dual-processor example.

Single-Processor Example

Figure 1 shows a hypothetical example of thread interruption on a single-processor system. For the sake of simplicity, the example omits the thread scheduler, clock interrupts, and so forth.

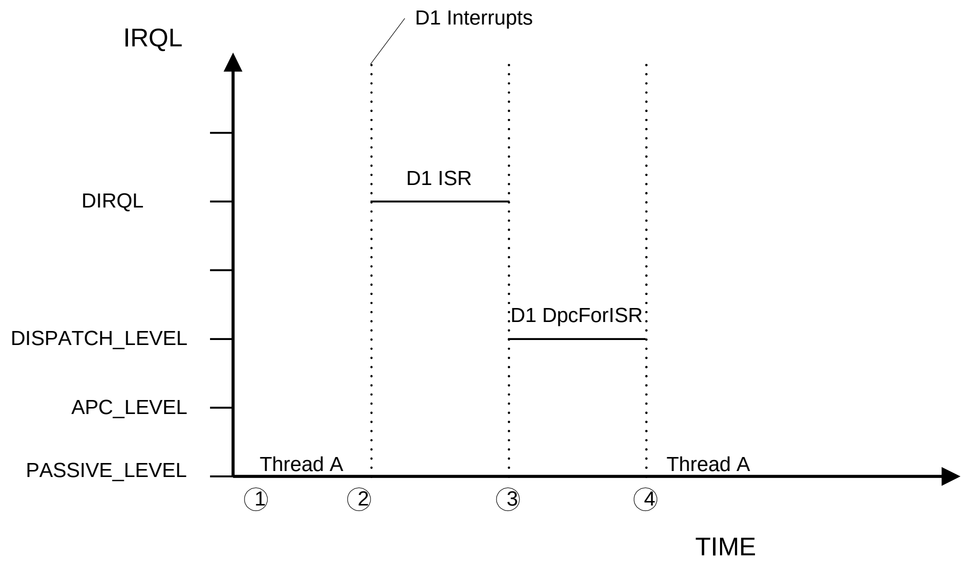

Figure 1. Thread Interruption on a Single-Processor System

The figure shows how thread execution proceeds over time, as follows:

1. Thread A is running at IRQL PASSIVE_LEVEL.
2. Device 1 interrupts at DIRQL. Thread A is interrupted, even though its quantum has not yet expired. The system suspends Thread A and runs the *InterruptService* (ISR) routine for Device 1. The *InterruptService* routine stops

Device 1 from interrupting, saves any data it requires for further processing, queues a *DpcForISR* routine, and exits.

3. No additional interrupts are pending at DIRQL for any device. Because deferred procedure calls (DPCs) run at IRQL DISPATCH_LEVEL, any entries in the system's queue of DPC routines will run before Thread A can resume.

   In this case, the queue contains the *DpcForIsr* routine that was queued by the Device 1 *InterruptService* routine. Because no further interrupts occur at IRQL>DISPATCH_LEVEL, the *DpcForIsr* routine runs to completion.

4. After the *DpcForIsr* routine exits, the DPC queue is empty and no other higher-priority threads are ready to run. Therefore, the system resumes running Thread A, which continues until one of the following occurs:
   - Its quantum expires.
   - It enters a wait state.
   - It exits.
   - A higher-priority thread becomes ready to run.
   - A hardware interrupt occurs.
   - The thread queues a DPC or an APC.

There is no guarantee that Thread A will exhaust its quantum. A thread can be interrupted or pre-empted any number of times during its quantum.

Multiprocessor Example

Figure 2 shows a hypothetical example of thread interruption on a multiprocessor system. This example omits clock interrupts and so forth, but it shows the system's thread scheduler.

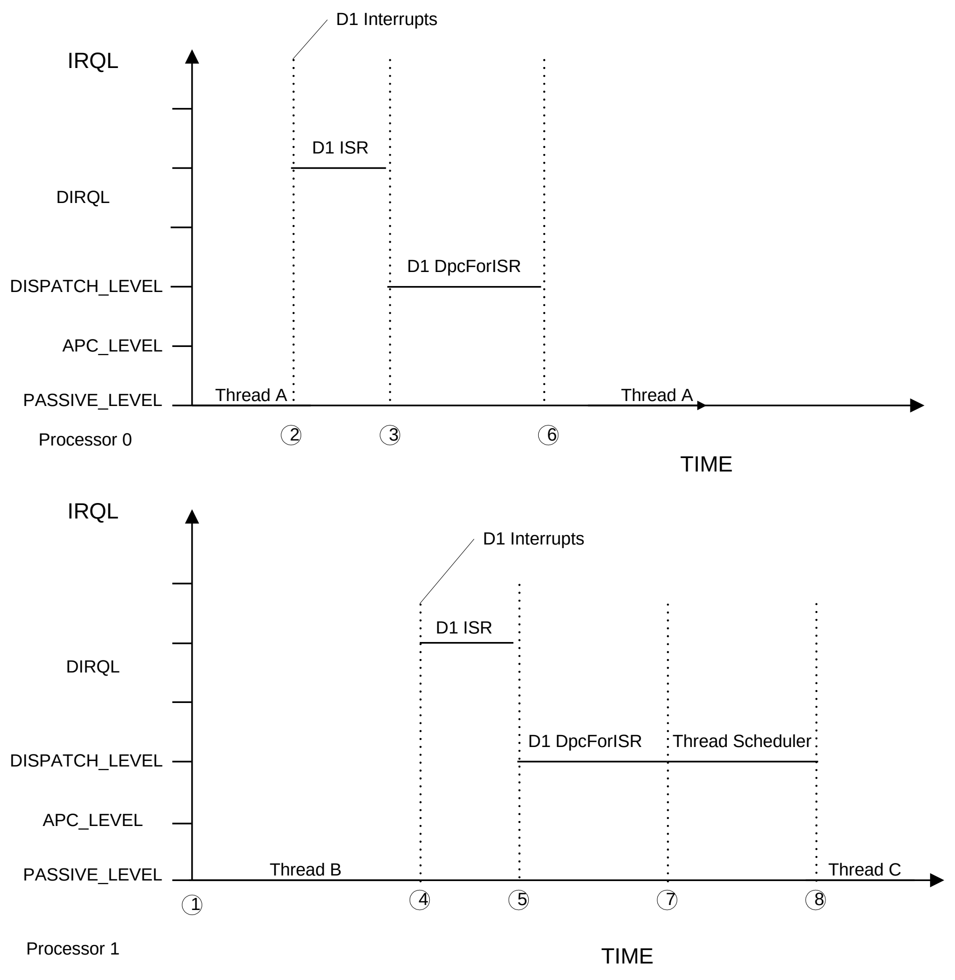

Figure 2. Thread Interruption on a Multiprocessor System

The figure shows thread execution on two processors, starting at the same time, as follows:

1. Processor 0 is running Thread A, while Processor 1 is running Thread B. Both threads run at IRQL=PASSIVE_LEVEL.
2. Device 1 interrupts on Processor 0, so the system raises IRQL on Processor 0 to DIRQL for Device 1 and runs the Device 1 *InterruptService* (ISR) routine. The Device 1 *InterruptService* routine queues a *DpcForIsr* routine. By default, the *DpcForIsr* routine is added to the queue for the same processor (Processor 0) on which the *InterruptService* routine is running.
3. Because the *DpcForIsr* routine runs on the same processor as the *InterruptService* routine, but at a lower IRQL, it does not start until after the *InterruptService* routine exits.
4. Device 1 interrupts again—this time on Processor 1—so the system raises IRQL on Processor 1 to DIRQL for Device 1 and runs the Device 1 *InterruptService* (ISR) routine on Processor 1. The Device 1 *InterruptService* routine queues a *DpcForIsr* routine to Processor 1 (the default behavior). The *DpcForIsr* routine starts on Processor 1 after the *InterruptService* routine exits.

5. The same *DpcForIsr* routine is now running on two processors at the same time, in response to two different interrupts from the same device. The driver must use spin locks to protect shared memory that the routines might access. In this case, the *DpcForIsr* routine on Processor 0 acquires the lock first, so Processor 1 spins while waiting for the lock.
6. The *DpcForIsr* routine on Processor 0 releases the lock, completes its work, and exits. The system now runs the next ready thread on Processor 0—in this case, Thread A.
7. After the *DpcForIsr* routine on Processor 0 releases the lock, the *DpcForIsr* routine on Processor 1 acquires the lock and performs its tasks. After it exits, the system's thread scheduling code runs to determine which thread to run next.
8. When the thread scheduler exits, no additional DPCs have been queued, so the highest-priority ready thread (Thread C) runs on Processor 1.

Testing for IRQL Problems

Errors related to IRQL are common and often result in a system crash with the bug check code IRQL_ NOT_LESS_OR_EQUAL. The Windows DDK includes several features that can help you to determine the IRQL at which driver code runs and to find IRQL-related problems:

- Routines and debugger commands that return the current IRQL
- The PAGED_CODE macro
- Forced IRQL checking in Driver Verifier

Techniques for Finding the Current IRQL

You can get the IRQL at which a processor is currently operating in two ways:

- Call the **KeGetCurrentIrql** routine.
- Use the !irql kernel-mode debugger extension command.

A driver can call **KeGetCurrentIrql** to get the IRQL at which the current processor is operating. This routine can be called from any IRQL. A driver can also determine whether it is operating in a critical region by calling **KeAreApcsDisabled** (available on Windows XP and later releases).

During debugging, you can use the !irql kernel-mode debugger extension command to find the IRQL at which a processor is operating. This command returns the IRQL at which the processor was operating immediately before the debugger became active. By default, the command returns the IRQL for the current processor, but you can also specify a processor number as a parameter. This command is available on Windows Server™ 2003 and later releases.

PAGED_CODE Macro

The PAGED_CODE macro can help you find IRQL problems that are related to page faults. If the processor is running at or above DISPATCH_LEVEL when the macro is called, the system will ASSERT. By placing the macro at the beginning of each driver routine that contains or calls pageable code, you can determine whether the routine performs actions that are invalid at the IRQL at which it is called.

However, the macro cannot help you find IRQL problems in code that subsequently raises IRQL. For example, if the processor is running at PASSIVE_LEVEL when the PAGED_CODE macro runs, but the routine later calls **KeAcquireSpinLock** (which raises IRQL to DISPATCH_LEVEL), the macro does not cause the system to ASSERT. You must use the Windows tool Driver Verifier to find such errors.

Driver Verifier Options

Driver Verifier (verifier.exe) performs numerous checks related to IRQL. By default, Driver Verifier checks for certain errors in calls to **KeRaiseIrql** and **KeLowerIrql**, memory allocation at an invalid IRQL, and acquiring or releasing spin locks at an invalid IRQL.

In addition, Driver Verifier also can perform forced IRQL checking. When you choose this option, Driver Verifier marks all pageable code and data (including the system's pageable memory pool, code, and data) whenever the driver requests a spin lock, calls **KeSynchronizeExecution**, or raises the IRQL to DISPATCH_LEVEL or higher. If the driver tries to access any of the pageable memory, Driver Verifier issues a bug check.

When forced IRQL checking is enabled, Driver Verifier gathers IRQL-related statistics, including the number of times the driver raised IRQL, acquired a spin lock, or called **KeSynchronizeExecution**. It also counts the number of times that the operating system paged the contents of memory to disk. Driver Verifier stores these statistics in global counters. You can display their values by using the Driver Verifier command line or graphical user interface, or by using the !verifier extension in the debugger. (The Driver Verifier command syntax depends on the version of Windows that is installed; see the Windows DDK for details.) Forced IRQL checking is not available for graphics drivers.

If your driver performs DMA, you should also use Driver Verifier's DMA verification option. This option checks for calls made to DMA routines at the wrong IRQL.

Best Practices for Drivers

To avoid problems related to thread context and IRQL, adopt these practices:

- Unless you are certain that a driver routine is called in a particular thread context, never make any assumptions about the contents of the user-mode address space.
- Know what driver routines can be called at IRQL>=DISPATCH_LEVEL and understand the restrictions that running at this level places on driver code.
- Store any data that can be accessed at IRQL>=DISPATCH_LEVEL in nonpaged memory. Possible locations are the device extension, a driver-allocated space in nonpaged pool memory, or the kernel-mode stack.
- Use Driver Verifier, the PAGED_CODE macro, and debugger extensions to help find IRQL-related bugs in drivers.
- Test drivers on as many hardware configurations as possible.

Call to Action and Resources

- Follow the best practices outlined in this paper.
- For more information about IRQL issues for drivers, see the companion paper "Locks, Deadlocks, and Synchronization," at http://www.microsoft.com/whdc/hwdev/driver/LOCKS.mspx.
- For additional information about Windows driver development, see Windows Hardware and Driver Central at http://www.microsoft.com/whdc.