Contents

Kernel-Mode Driver Architecture Design Guide

6/25/2019 • 2 minutes to read • Edit Online

This section includes general concepts to help you understand kernel-mode programming and describes specific techniques of kernel programming. This section is divided into four parts:

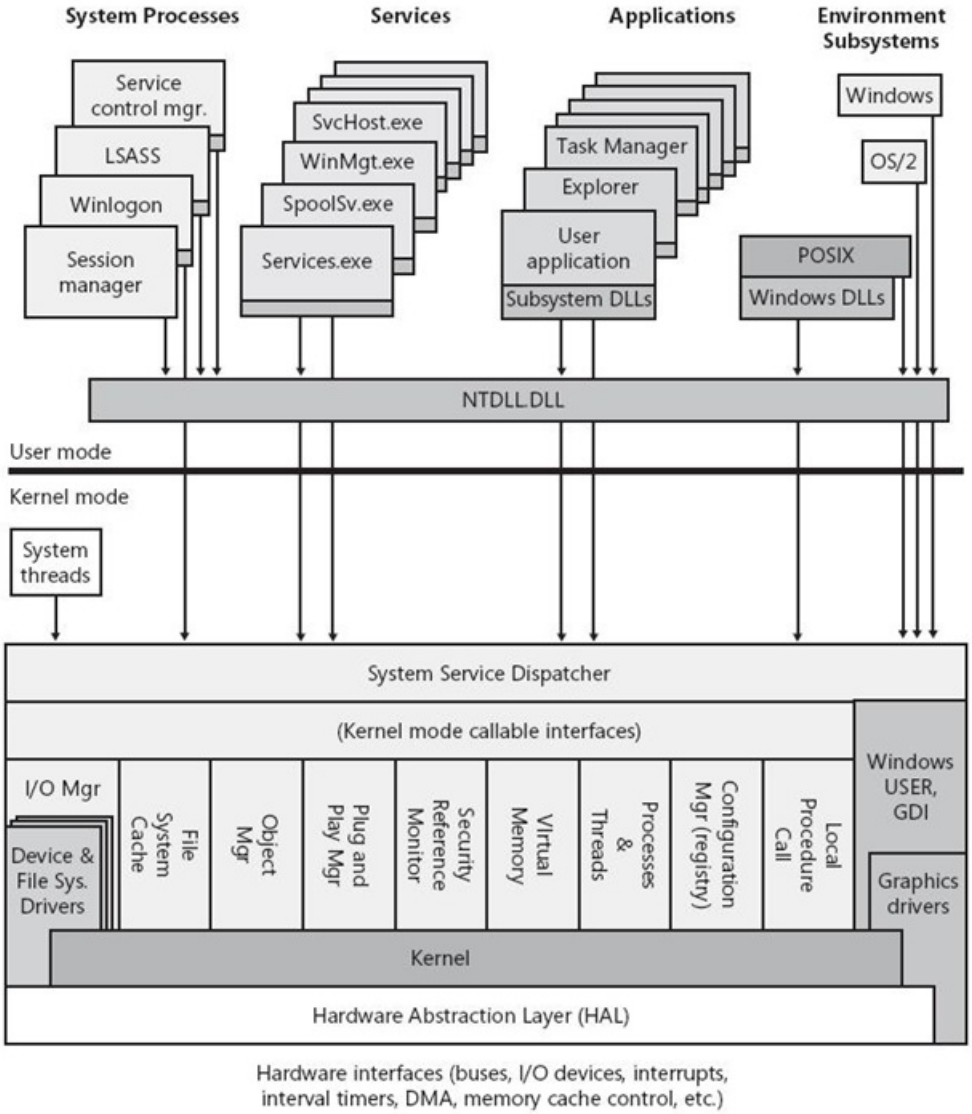
- Introduction to Windows Drivers provides a general overview of Windows components, lists the types of device drivers used in Windows, discusses the goals of Windows device drivers, and discusses generic sample device drivers included in the kit.
- Kernel-Mode Managers and Libraries lists the primary kernel-mode components of the Windows operating system.
- Writing WDM Drivers provides information needed to write drivers using the Windows Driver Model (WDM).
- Driver Programming Techniques describes techniques that you can use to program Windows kernel-mode device drivers.

  Note For information about programming interfaces that your driver can implement or call, see Kernel-Mode Driver Reference.

Overview of Windows Components

10/7/2019 • 2 minutes to read • Edit Online

The following figure shows the major internal components of the Windows operating system.



As the figure shows, the Windows operating system includes both user-mode and kernel-mode components. For more information about Windows user and kernel modes, see User Mode and Kernel Mode.

Drivers call routines that are exported by various kernel components. For example, to create a device object, you would call the **IoCreateDevice** routine which is exported by the I/O manager. For a list of kernel-mode routines that drivers can call, see Driver Support Routines.

In addition, drivers must respond to specific calls from the operating system and can respond to other system calls. For a list of kernel mode routines that drivers may need to support, see Standard Driver Routines.

Not all kernel-mode components are pictured in the figure above. For a list of kernel mode components, see Kernel-Mode Managers and Libraries.

Types of Windows Drivers

10/7/2019 • 2 minutes to read • Edit Online

There are two basic types of Microsoft Windows drivers:

- *User-mode drivers* execute in user mode, and they typically provide an interface between a Win32 application and kernel-mode drivers or other operating system components.

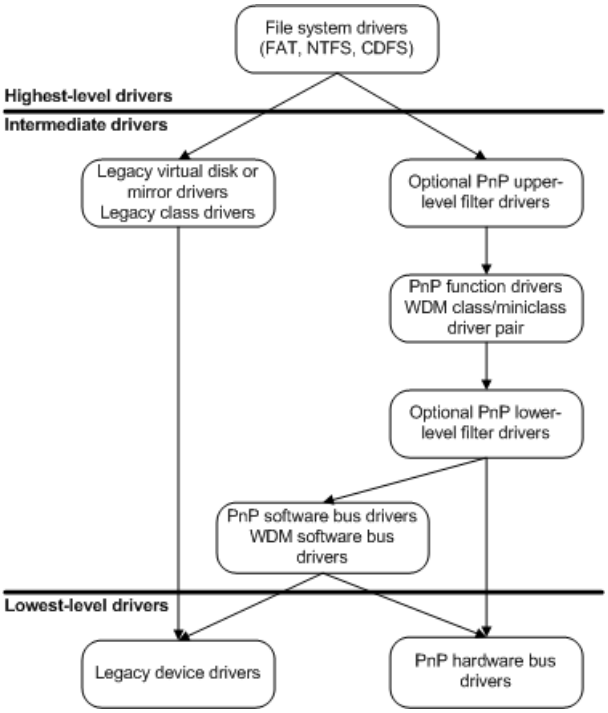
  For example, in Windows Vista, all printer drivers execute in user mode. For more information about printer driver components, see Introduction to Printing.

- *Kernel-mode drivers* execute in kernel mode as part of the executive, which consists of kernel-mode operating system components that manage I/O, Plug and Play memory, processes and threads, security, and so on. Kernel-mode drivers are typically layered. Generally, higher-level drivers typically receive data from applications, filter the data, and pass it to a lower-level driver that supports device functionality.

  Some kernel-mode drivers are also *WDM drivers*, which conform to the Windows Driver Model (WDM). All WDM drivers support Plug and Play, and power management. WDM drivers are source-compatible (but not binary-compatible) across Windows 98/Me and Windows 2000 and later operating systems.

  Like the operating system itself, kernel-mode drivers are implemented as discrete, modular components that have a well-defined set of required functionalities. All kernel-mode drivers supply a set of system-defined standard driver routines.

The following figure divides kernel-mode drivers into several types.



As shown in the figure, there are three basic types of kernel-mode drivers in a driver stack: highest-level, intermediate, and lowest-level. Each type differs only slightly in structure but greatly in functionality:

1. *Highest-level drivers*. Highest-level drivers include file system drivers (FSDs) that support file systems, such as:

   - NTFS

- File allocation table (FAT)
- CD-ROM file system (CDFS)

Highest-level drivers always depend on support from underlying lower-level drivers, such as intermediate-level function drivers and lowest-level hardware bus drivers.

2. *Intermediate drivers*, such as a virtual disk, mirror, or device-type-specific *class driver*. Intermediate drivers depend on support from underlying lower-level drivers. Intermediate drivers are subdivided further as follows:

   - *Function drivers* control specific peripheral devices on an I/O bus.
   - *Filter drivers* insert themselves above or below function drivers.
   - *Software bus drivers* present a set of child devices to which still higher-level class, function, or filter drivers can attach themselves.

     For example, a driver that controls a multifunction adapter with an on-board set of heterogeneous devices is a software bus driver.
   - Any system-supplied *class driver* that exports a system-defined class/miniclass interface is, in effect, an intermediate driver with one or more linked *miniclass drivers* (sometimes called *minidrivers*). Each linked class/minidriver pair provides functionality that is equivalent to that of a function driver or a software bus driver.

3. *Lowest-level drivers* control an I/O bus to which peripheral devices are connected. Lowest-level drivers do not depend on lower-level drivers.

   - Hardware *bus drivers* are system-supplied and usually control dynamically configurable I/O buses.

     Hardware bus drivers work with the Plug and Play manager to configure and reconfigure system hardware resources, for all child devices that are connected to the I/O buses that the driver controls. These hardware resources include mappings for device memory and interrupt requests (IRQs). (Hardware bus drivers subsume some of the functionality that the HAL component provided in releases of the Windows NT-based operating system earlier than Windows 2000.)
   - *Legacy drivers* that directly control a physical device are lowest-level drivers.

Design Goals for Kernel-Mode Drivers

12/5/2018 • 2 minutes to read • Edit Online

Kernel-mode drivers share many of the design goals of the operating system, particularly those of the system I/O manager. Kernel-mode drivers are designed to be:

- Portable from one platform to another.
- Configurable to various hardware and software platforms.
- Always preemptible and always interruptible.
- Multiprocessor-safe on multiprocessor platforms.
- Object-based.
- Packet-driven I/O with reusable IRPs.
- Capable of supporting asynchronous I/O.

Portable

6/25/2019 • 2 minutes to read • Edit Online

All drivers must be portable across all Windows-supported hardware platforms. To achieve cross-platform portability, driver writers should:

- Code in C (no assembly language).
- Interact with Windows by only using the programming interfaces and headers that are supplied in the WDK.

Coding Drivers in C

All kernel-mode drivers should be written in C so that they can be recompiled with a system-compatible C compiler, relinked, and run on different Microsoft Windows platforms without rewriting or replacing any code. Most operating system components are coded entirely in C, with only small pieces of the HAL and kernel components written in assembly language, so that the operating system is readily portable across hardware platforms. You cannot use many C++ language constructs in kernel-mode drivers, so you should carefully evaluate using such constructs. For more information about issues that arise when drivers include C++ features, see the C++ for Kernel Mode Drivers: Pros and Cons white paper.

Drivers should not rely on the features of any particular system-compatible C compiler or C support library if those features are not guaranteed to be supported by other system-compatible compilers. In general, driver code should conform to the ANSI C standard and not depend on anything that this standard describes as "implementation-defined."

To write portable drivers, it is best to avoid:

- Dependencies on data types that can vary in size or layout from one platform to another.
- Calling any standard C runtime library function that maintains state.
- Calling any standard C runtime library function for which the operating system provides an alternative support routine.

Using WDK-Supplied Interfaces

Each Windows NT executive component exports a set of kernel-mode driver support routines that drivers and all other kernel-mode components call. If the underlying implementation of a support routine changes over time, its callers remain portable because the interface to the defining component does not change.

The WDK supplies a set of header files that define system-specific data types and constants that drivers (and all other kernel-mode components) use to help maintain portability from one platform to another. All kernel-mode drivers include one of the master WDK kernel-mode header files, Wdm.h or Ntddk.h. The master header files pull in not only system-supplied headers that define the basic kernel-mode types, but also appropriate selections from any processor-architecture-specific headers when a driver is compiled with the corresponding compiler directive.

Some drivers, such as SCSI miniport drivers, NDIS drivers, and video miniport drivers, include other system-supplied header files.

If a driver requires platform-dependent definitions, it is best to isolate those definitions within **#ifdef** statements, so that each driver can be compiled and linked for the appropriate hardware platform. However, you can almost always avoid implementing any platform-specific, conditionally compiled code in a driver by using the support routines, macros, constants, and types that the WDK master header files provide.

Kernel-mode drivers can use kernel-mode **Rtl*Xxx*** routines that are documented in the WDK. Kernel-mode drivers

cannot call user-mode **Rtl*Xxx*** routines.

Configurable

12/5/2018 • 2 minutes to read • Edit Online

Today's peripheral devices must be *hardware-configurable*, and their drivers must be *software-configurable*.

A device is hardware-configurable if it can accept different assignments of the system's hardware resources, such as I/O port numbers, without being physically modified. For example, if a set of hot-pluggable Plug and Play disks are connected in a redundant array of independent disks (RAID) configuration, a user can swap disks while the system is running. If a device is hardware-configurable, its drivers cannot contain hard-coded, system-dependent values for the device's hardware resources.

A driver is software-configurable if:

- It can receive and change its device's hardware resources dynamically.

  Drivers that support Plug and Play do not contain hard-coded values for a device's hardware resources, nor does the driver poll the device to determine its resource assignments. Instead, the system dynamically assigns resources to the device, and then supplies resource values to the driver.

- It was written with no assumptions about other drivers that might reside above or below it in its driver stack.

  For example, the design of a lower-level device driver for a disk must be flexible enough to support multiple file systems that are implemented by multiple high-level file system drivers, possibly on a single computer.

  Additionally, if a computer has sufficient mass storage capacity, that same lower-level disk driver must not interfere with an intermediate driver's support for fault tolerance (implemented as mirrored partitions, stripe sets, or volume sets) within a file system.

The PnP manager and each PnP hardware bus driver work together to provide an interface between the platform's hardware for a specific type of I/O bus and the system's software. The PnP manager builds a device tree, with nodes that represent all the devices on the system, including buses. For each device, the PnP manager maintains two lists:

- A list of the hardware resources that the device is capable of using.
- A list of the hardware resources that are actually assigned to the device.

Device drivers assist the PnP manager in creating these lists, which are maintained in the registry. As devices are added to and removed from the system, the PnP manager reassigns resources as necessary and updates the lists.

The system's hardware abstraction layer (HAL) component, which is implemented as a dynamic-link library, is responsible for some of the hardware-level, platform-specific support that is needed by other system components, including kernel-mode drivers.

Always Preemptible and Always Interruptible

2/14/2019 • 3 minutes to read • Edit Online

The goal of the preemptible, interruptible design of the operating system is to maximize system performance. Any thread can be preempted by a thread with a higher priority, and any driver's interrupt service routine (ISR) can be interrupted by a routine that runs at a higher interrupt request level (IRQL).

The kernel component determines when a code sequence runs, according to one of these prioritizing criteria:

- The kernel-defined run-time priority scheme for threads.

  Every thread in the system has an associated priority attribute. In general, most threads have *variable* priority attributes: they are always preemptible and are scheduled to run round-robin with all other threads that are currently at the same priority level. Some threads have *real-time* priority attributes: these time-critical threads run to completion unless they are preempted by a thread that has a higher real-time priority attribute. The Microsoft Windows architecture does not provide an inherently real-time system.

  Whatever its priority attribute, any thread in the system can be preempted when hardware interrupts and certain types of software interrupts occur.

- The kernel-defined *interrupt request level* (IRQL) to which a particular interrupt vector is assigned on a given platform.

  The kernel prioritizes hardware and software interrupts so that some kernel-mode code, including most drivers, runs at higher IRQLs, thereby making it have a higher scheduling priority than other threads in the system. The particular IRQL at which a piece of kernel-mode driver code executes is determined by the *hardware priority* of its underlying device.

  Kernel-mode code is always interruptible: an interrupt with a higher IRQL value can occur at any time, thereby causing another piece of kernel-mode code that has a higher system-assigned IRQL to be run immediately on that processor. However, when a piece of code runs at a given IRQL, the kernel masks all interrupt vectors with a lesser or equal IRQL value on the processor.

The lowest IRQL level is called PASSIVE_LEVEL. At this level, no interrupt vectors are masked. Threads generally run at IRQL=PASSIVE_LEVEL. The next higher IRQL levels are for software interrupts. These levels include APC_LEVEL, DISPATCH_LEVEL or, for kernel debugging, WAKE_LEVEL. Device interrupts have still higher IRQL values. The kernel reserves the highest IRQL values for system-critical interrupts, such as those from the system clock or bus errors.

Some system support routines run at IRQL=PASSIVE_LEVEL, either because they are implemented as pageable code or access pageable data, or because some kernel-mode components set up their own threads.

Similarly, some standard driver routines usually run at IRQL=PASSIVE_LEVEL. However, several standard driver routines run either at IRQL=DISPATCH_LEVEL or, for a lowest-level driver, at device IRQL (also called *DIRQL*). For more information about IRQLs, see Managing Hardware Priorities.

Every routine in a driver is interruptible. This includes any routine that is running at a higher IRQL than PASSIVE_LEVEL. Any routine that is running at a particular IRQL retains control of the processor only if no interrupt for a higher IRQL occurs while that routine is running.

Unlike the drivers in some older personal computer operating systems, a Microsoft Windows driver's ISR is never a large, complex routine that does most of the driver's I/O processing. This is because any driver's *interrupt service routine* (ISR) can be interrupted by another routine (for example, by another driver's ISR) that runs at a higher IRQL. Thus, the driver's ISR does not necessarily retain control of a CPU, uninterrupted, from the beginning of its

execution path to the end.

In Windows drivers, an ISR typically saves hardware state information, queues a *deferred procedure call* (DPC), and then quickly exits. Later, the system dequeues the driver's DPC so that the driver can complete I/O operations at a lower IRQL (DISPATCH_LEVEL). For good overall system performance, all routines that run at high IRQLs must relinquish control of the CPU quickly.

In Windows, all threads have a thread context. This context consists of information that identifies the process that owns the thread, plus other characteristics such as the thread's access rights.

In general, only a highest-level driver is called in the context of the thread that is requesting the driver's current I/O operation. An intermediate-level or lowest-level driver can never assume that it is executing in the context of the thread that requested its current I/O operation.

Consequently, driver routines usually execute in an *arbitrary thread context*—the context of whatever thread is current when a standard driver routine is called. For performance reasons (to avoid context switches), very few drivers set up their own threads.

Multiprocessor-Safe

1/11/2019 • 3 minutes to read • Edit Online

The Microsoft Windows NT-based operating system is designed to run uniformly on uniprocessor and symmetric multiprocessor (SMP) platforms, and kernel-mode drivers should be designed to do likewise.

In any Windows multiprocessor platform, the following conditions exist:

- All CPUs are identical, and either all or none of the processors must have identical coprocessors.
- All CPUs share memory and have uniform access to memory.
- In a *symmetric* platform, every CPU can access memory, take an interrupt, and access I/O control registers. (By contrast, in an *asymmetric* multiprocessor machine, one CPU takes all interrupts for a set of subordinate CPUs.)

To run safely on an SMP platform, an operating system must guarantee that code that executes on one processor does not simultaneously access and modify data that another processor is accessing and modifying. For example, if a lowest-level driver's ISR is handling a device interrupt on one processor, it must have exclusive access to device registers or critical, driver-defined data, in case its device interrupts simultaneously on another processor.

Furthermore, drivers' I/O operations that are serialized in a uniprocessor machine can be overlapped in an SMP machine. That is, a driver's routine that processes incoming I/O requests can be executing on one processor while another routine that communicates with the device executes concurrently on another processor. Whether kernel-mode drivers are executing on a uniprocessor or symmetric multiprocessor machine, they must synchronize access to any driver-defined data or system-provided resources that are shared among driver routines, and synchronize access to the physical device, if any.

The Windows NT kernel component exports a synchronization mechanism, called a spin lock, that drivers can use to protect shared data (or device registers) from simultaneous access by one or more routines that are running concurrently on a symmetric multiprocessor platform. The kernel enforces two policies regarding the use of spin locks:

- Only one routine can hold a particular spin lock at any given moment. Before accessing shared data, each routine that must reference the data must first attempt to acquire the data's spin lock. To access the same data, another routine must acquire the spin lock, but the spin lock cannot be acquired until the current holder releases it.
- The kernel assigns an IRQL value to each spin lock in the system. A kernel-mode routine can acquire a particular spin lock only when the routine is run at the spin lock's assigned IRQL.

These policies prevent a driver routine that usually runs at a lower IRQL but currently holds a spin lock from being preempted by a higher-priority driver routine that is trying to acquire the same spin lock. Thus, a deadlock is avoided.

The IRQL that is assigned to a spin lock is generally that of the highest-IRQL routine that can acquire the spin lock.

For example, a lowest-level driver's ISR frequently shares a state area with the driver's DPC routine. The DPC routine calls a driver-supplied *critical section* routine to access the shared area. The spin lock that protects the shared area has an IRQL equal to the DIRQL at which the device interrupts. As long as the critical-section routine holds the spin lock and accesses the shared area at DIRQL, the ISR cannot be run in either a uniprocessor or SMP machine.

- The ISR cannot be run in a uniprocessor machine because the device interrupt is masked, as described in

Always Preemptible and Always Interruptible.

- In an SMP machine, the ISR cannot acquire the spin lock that protects the shared data while the critical-section routine holds the spin lock and accesses the shared data at DIRQL.

A set of kernel-mode threads can synchronize access to shared data or resources by waiting for one of the kernel's dispatcher objects: an event, mutex, semaphore, timer, or another thread. However, most drivers do not set up their own threads because they have better performance when they avoid thread-context switches. Whenever time-critical kernel-mode support routines and drivers run at IRQL = DISPATCH_LEVEL or at DIRQL, they must use the kernel's spin locks to synchronize access to shared data or resources.

For more information, see Spin Locks, Managing Hardware Priorities, and Kernel Dispatcher Objects.

Object-Based

6/25/2019 • 2 minutes to read • Edit Online

The Microsoft Windows NT-based operating system is object-based. Various components in the executive define one or more object types. Each component exports kernel-mode support routines that manipulate instances of its object types. No component can directly access another component's objects. To use another component's objects, a component must call the exported support routines.

This design allows the operating system to be both portable and flexible. For example, it is possible for a future release of the operating system to contain a recoded kernel component that defines the same object types, but with entirely different internal structures. If this hypothetical recoded version of the kernel exports a set of support routines that have the same names and parameters as the existing set, the internal changes would have no effect on the portability of any other executive component in the existing system.

Likewise, to remain portable and configurable, drivers must communicate with the operating system and with each other by using only the support routines and other interfaces that are described in the WDK.

Like the operating system, drivers are also object-based. For example:

- *File objects* represent a user-mode application's connection to a device.
- *Device objects* represent each driver's logical, virtual, or physical devices.
- *Driver objects* represent each driver's load image.

The I/O manager defines the structure and interfaces for file objects, device objects, and driver objects.

Like any other executive component, drivers use objects by calling kernel-mode support routines that the I/O manager and other system components export. Kernel-mode support routines generally have names that identify the specific object that each routine manipulates and the operation that each routine performs on that object. These support routine names have the following form:

PrefixOperationObject

where

Prefix Identifies the kernel-mode component that exports the support routine and, usually, the component that defined the object type. Most prefixes have two letters.

Operation Describes what is done to the object.

Object Identifies the type of object.

For example, the I/O manager's **IoCreateDevice** routine creates a device object to represent a physical, logical, or virtual device as the target of I/O requests.

One system component can export routines that call another component's support routines. This can reduce the number of calls that a driver must make. The I/O manager, in particular, exports certain routines that make it easier to develop drivers. For example, **IoConnectInterruptEx**, calls the kernel support routines for *interrupt objects*.

Object Opacity

Some system-defined objects are *opaque*: only the defining system component is aware of such an object's internal structure and can directly access all of the data that an object contains. The system component that defines an opaque object exports support routines that drivers and other kernel-mode components can call to manipulate that object. Drivers never access opaque object structures directly.

Note To maintain driver portability, drivers must use the system-supplied support routines to manipulate system-defined objects. The defining system component can change the internal structure of its object types at any time.

Packet-Driven I/O with Reusable IRPs

12/5/2018 • 2 minutes to read • Edit Online

The I/O manager, Plug and Play manager, and power manager use*I/O request packets* (IRPs) to communicate with kernel-mode drivers, and to allow drivers to communicate with each other.

The I/O manager performs the following steps:

- Accepts I/O requests, which usually originate from user-mode applications.
- Creates IRPs to represent the I/O requests.
- Routes the IRPs to the appropriate drivers.
- Tracks the IRPs until they are completed.
- Returns the status to the original requester of each I/O operation.

An IRP might be routed to more than one driver. For example, a request to open a file on a disk might go first to a file system driver, through an intermediate mirror driver, and ultimately to a disk driver and, possibly, to a PnP hardware bus driver. This set of drivers is known as a *driver stack*.

Therefore, each IRP has a *fixed part*, plus one driver-specific *I/O stack location* for each driver that controls the device:

- In the fixed part (or *header*), the I/O manager maintains information about the original request, such as the caller's thread ID and parameters, the address of the device object on which a file is open, and so forth. The fixed part also contains an *I/O status block*, in which drivers set information about the status of the requested I/O operation.
- In the highest-level driver's I/O stack location, the I/O manager, Plug and Play manager, or power manager sets driver-specific parameters, such as the function code of the requested operation and the context that the corresponding driver uses to determine what it should do. In turn, each driver sets up the I/O stack location of the next-lower driver in the driver stack.

As each driver processes an IRP, it can access its I/O stack location in the IRP, thereby reusing the IRP at each stage of the driver's operations. In addition, higher-level drivers can create (or reuse) IRPs to send requests down to even lower-level drivers.

For a detailed discussion of IRPs, see Handling IRPs.

Supporting Asynchronous I/O

12/5/2018 • 2 minutes to read • Edit Online

The I/O manager provides asynchronous I/O support so that the originator of an I/O request (usually a user-mode application but sometimes another driver) can continue executing, rather than wait for its I/O request to be completed. Asynchronous I/O support improves both the overall system throughput and the performance of any code that makes an I/O request.

With asynchronous I/O support, kernel-mode drivers do not necessarily process I/O requests in the same order in which they were sent to the I/O manager. The I/O manager, or a higher-level driver, can reorder I/O requests as they are received. A driver can split a large data transfer request into smaller transfer requests. Moreover, a driver can overlap I/O request processing, particularly in a symmetric multiprocessor platform, as mentioned in Multiprocessor-Safe.

Furthermore, a kernel-mode driver's processing of an individual I/O request is not necessarily serialized. That is, a driver does not necessarily process each IRP to completion before it starts processing the next incoming I/O request.

When a driver receives an IRP, it responds by carrying out as much IRP-specific processing as it can. If the driver supports asynchronous IRP processing, it can send an IRP to the next driver, if necessary, and begin processing the next IRP without waiting for the first one to be completed. The driver can register a "completion routine," which the I/O manager calls when another driver has finished processing an IRP. Drivers provide a status value in the IRP's I/O status block, which other drivers can access to determine the status of an I/O request.

Drivers can maintain state information about their current I/O operations in a special part of their device objects, called a device extension.

For more information, see Handling IRPs and Input/Output Techniques.

Sample Kernel-Mode Drivers

6/25/2019 • 2 minutes to read • Edit Online

The WDK provides various sample kernel-mode drivers. After you have installed the WDK, the `src\general` subdirectory contains sample driver code that is applicable to all kernel-mode drivers. The samples are also maintained online. These samples include the following:

DCHU

Applies the DCHU design principles (Declarative, Componentized, Hardware Support Apps [HSA], and Universal API compliance). You can use it as a model for your own universal driver package.

PLX9x5x

This sample demonstrates how to write driver for a generic PCI device using Windows Driver Framework.

SimpleMediaSource

This sample demonstrates how to create a custom media source and driver package that can be installed as a Camera and produce frames.

SystemDma/wdm

This sample demonstrates the usage of V3 System DMA. It shows how a driver could use a system DMA controller supported by Windows to write data to a hardware location using DMA.

WinHEC 2017 Lab

WinHEC 2017/Optimizing Windows Performance

cancel

Demonstrates the use of cancel-safe IRP queues.

echo

event

Demonstrates techniques that kernel-mode drivers can use to notify applications of hardware events, if the application requests notification. One technique uses event objects and the other relies on queuing the notification request until an event occurs.

filehistory

The FileHistory sample is a console application that starts the file history service, if it is stopped, and schedules regular backups. The application requires, as a command-line parameter, the path name of a storage device to use as the default backup target.

IOCTL sample

Demonstrates how drivers should support I/O control codes.

obcallback

The ObCallback sample driver demonstrates the use of registered callbacks for process protection. The driver registers control callbacks which are called at process creation.

pcidrv

This sample demonstrates how to write a KMDF driver for a PCI device. The sample works with the Intel 82557/82558 based PCI Ethernet Adapter (10/100) and Intel compatibles.

perfcounters/kcs

The Kcs sample driver demonstrates the use of the kernel-mode performance library.

registry/regfltr

The RegFltr sample shows how to write a registry filter driver.

toaster

Provides sample code for a set of drivers that conform to the Windows Driver Model (WDM). This sample also includes sample installation software.

tracedrv

Shows how to use WPP software tracing.

UMDF Driver Skeleton Sample

This sample demonstrates how to use version 1 of the User-Mode Driver Framework to write a minimal driver.

Firefly KMDF filter driver for a HID device Along with illustrating how to write a filter driver, this sample shows how to use remote I/O target interfaces to open a HID collection in kernel-mode and send IOCTL requests to set and get feature reports, as well as how an application can use WMI interfaces to send commands to a filter driver.

Other subdirectories of the `\src` directory contain sample code for kernel-mode drivers for various types of hardware.

See also

Microsoft Windows driver samples on GitHub

Surface Team Driver Development Best Practices

8/21/2019 • 10 minutes to read • Edit Online

Introduction

These driver development guidelines were developed over many years by driver developers at Microsoft. Over time when drivers misbehaved and lessons were learned, those lessons were captured and evolved to be this set of guidance. These best practices are used by the Microsoft Surface Hardware team to develop and maintain the device driver code that support the unique Surface hardware experiences.

Like any set of guidelines, there will be legitimate exceptions and alternative approaches that will be equally valid. Consider incorporating these guidelines into your development standards or using them to start your domain specific guidelines for your development environment and your unique requirements.

Common mistakes made by driver developers

Handling I/O

1. Accessing buffers retrieved from IOCTLs without validating the length. See Failure to Check the Size of Buffers.
2. Performing blocking I/O in the context of a user thread or random thread context. See Introduction to Kernel Dispatcher Objects.
3. Sending synchronous I/O to another driver without timeout. See Sending I/O Requests Synchronously.
4. Using neither-io IOCTLs without understanding security implications. See Using Neither Buffered Nor Direct I/O.
5. Not checking the return status of WdfRequestForwardToIoQueue or not handling failure correctly and resulting in abandoned WDFREQUESTs.
6. Keeping the WDFREQUEST outside the queue in a non-cancelable state. See Managing I/O Queues, Completing I/O Requests and Canceling I/O Requests.
7. Trying to manage cancelation using Mark/UnmarkCancelable function instead of using IoQueues. See Framework Queue Objects.
8. Not knowing the difference between file handle Cleanup and Close operations. See Errors in Handling Cleanup and Close Operations.
9. Overlooking potential recursions with I/O completion and resubmission from the completion routine.
10. Not being explicit about the power management attributes of WDFQUEUEs. Not documenting the power management choice clearly. This is the primary cause of Bug Check 0x9F: DRIVER_POWER_STATE_FAILURE in WDF drivers. When the device is removed, the framework purges IO from the power managed queue and non-power managed queue in different stages of removal process. Non power managed queues are purged when the final IRP_MN_REMOVE_DEVICE is received. So if you are holding I/O in an non-power managed queue, it's a good practice to explicitly purges the I/O in the context of EvtDeviceSelfManagedIoFlush to avoid deadlock.
11. Not following the rules of handling IRPs. See Errors in Handling Cleanup and Close Operations.

Synchronization

1. Holding locks for code that doesn't need protection. Do not hold a lock for an entire function when only a small number of operations needs to be protected.
2. Calling out of drivers with locks held. This is the primary causes of deadlocks.
3. Using interlocked primitives to create a locking scheme instead of using appropriate system provided locking primitives such as mutex, semaphore and spinlocks. See Introduction to Mutex Objects, Semaphore Objects and Introduction to Spin Locks.

4. Using a spinlock where some type of passive lock would be more appropriate. See Fast Mutexes and Guarded Mutexes and Event Objects. For additional perspective on locks review the OSR Article - The State of Synchronization.
5. Opting into WDF synchronization and execution level model without full understanding of implications. See Using Framework Locks. Unless your driver is monolithic top-level driver directly interacting with the hardware, avoid opting into WDF synchronization as it can lead to deadlocks due to recursion.
6. Acquiring KEVENT, Semaphore, ERESOURCE, UnsafeFastMutex in the context of multiple threads without entering critical region. Doing this can lead to DOS attack because a thread holding one of these locks can be suspended. See Synchronization Techniques.
7. Allocating KEVENT on thread stack and returning to the caller while the EVENT is still in use. Typically done when used with IoBuildSyncronousFsdRequest or IoBuildDeviceIoControlRequest. Caller of these calls should make sure that they don't unwind from the stack until I/O manager has signaled the event when the IRP is completed.
8. Indefinitely waiting in dispatch routines. In general, any kind of wait in dispatch routine is a bad practice.
9. Inappropriately checking the validity of an object (if blah == NULL) before deleting it. This typically means the author doesn't have full understanding of the code that controls the lifetime of the object.

Object Management

1. Not explicitly parenting WDF objects. See Introduction to Framework Objects.
2. Parenting WDF object to WDFDRIVER instead of parenting to an object that provides better lifetime management and optimizes memory usage. For example, parenting WDFREQUEST to a WDFDEVICE instead of IOTARGET. See Using General Framework Objects, Framework Object Life Cycle and Summary of Framework Objects.
3. Not doing rundown protection of shared memory resources accessed across drivers. See ExInitializeRundownProtection function.
4. Mistakenly queuing the same work item while the previous one is already in the queue or already running. This is can be a problem if the client makes an assumption that every work item queued is going to get executed. See Using Framework WorkItems. For more information about queuing WorkItems, see the DMF_QueuedWorkitem module in the Driver Module Framework (DMF) project - https://github.com/Microsoft/DMF.
5. Queuing timer before posting the message the timer is expected to process. See Using Timers.
6. Performing an operation in a workitem that can block or take indefinitely long time to complete.
7. Designing a solution that results in a flood of work items to be queued. It can lead to unresponsive system or DOS attack if the bad guy can control the action (e.g. pumping I/O in to a driver that queues a new work item for every I/O). See Using Framework Work Items.
8. Not ensuing that work item DPC callbacks have run to completion before deleting the object. See Guidelines for Writing DPC Routines and the WdfDpcCancel function.
9. Creating threads instead of using work items for short duration/non-polling tasks. See System Worker Threads.
10. Not ensuring threads have run to completion before deleting or unload driver. For more information about thread rundown synchronization, look at the code associated with look at the code associated with DMF_Thread module in the Driver Module Framework (DMF) project - https://github.com/Microsoft/DMF.
11. Using a single driver to manage devices that are different but interdependent and using global variables to share information.

Memory

1. Not marking passive-execution code as PAGEABLE, when possible. Paging driver code can reduce the size of the driver's code footprint, thus freeing system space for other uses. Be cautious marking code pageable that raises IRQL >= DISPATCH_LEVEL or could be called at raised IRQL. See When Should Code and Data Be Pageable and Making Drivers Pageable and Detecting Code That Can Be Pageable.
2. Declaring large structures on the stack, Should use the heap/poolinstead. See Using the KernelStack and

Allocating System-Space Memory.

3. Unnecessarily zeroing WDF Object context. This can indicate a lack of clarity on when memory will be zeroed out automatically.

General Driver Guidelines

1. Mixing WDM and WDF primitives. Using WDM primitives where WDF primitives can be used. Using WDF primitives protects you from gotchas, improves debugging and more importantly makes your driver portable to usermode.
2. Naming FDOs and creating symbolic links when not needed. See Manage driver access control.
3. Copy pasting and using GUIDs and other constant values from sample drivers.
4. Consider the use of the Driver Module Framework (DMF) open source code in your driver project. DMF is an extension to WDF that enables extra functionality for a WDF driver developer. See Introducing Driver Module Framework.
5. Using registry as an inter-process notification mechanism or as a mailbox. For an alternative, see DMF_NotifyUserWithEvent and DMF_NotifyUserWithRequest modules available in the DMF project - https://github.com/Microsoft/DMF.
6. Assuming all parts of registry will be available for access during the early boot phase of the system.
7. Taking dependency on the load order of another driver or service. As the load order can be changed outside of the control of your driver, this can result in a driver that works initially, but later fails in an unpredictable pattern.
8. Recreating driver libraries that are already available, such as WDF provides for PnP described in Supporting PnP and Power Management in Your Driver or those provided in the bus interface as described in the OSR article Using Bus Interfaces for Driver to Driver Communication.

PnP/Power

1. Interfacing with another driver in a non-pnp friendly way - not registering for pnp device change notifications. See Registering for Device Interface Change Notification.
2. Creating ACPI nodes to enumerate devices and creating power dependencies among them instead of using bus driver or system provided software device creation interfaces to PNP and power dependencies in an elegant way. See Supporting PnP and Power Management in Function Drivers.
3. Marking the device not-disableable - forcing a reboot on driver update.
4. Hiding the device in the device manager. See Hiding Devices from Device Manager.
5. Making assumptions that driver will be used for only one instance of the device.
6. Making assumptions that driver will never get unloaded. See PnP Driver's Unload Routine.
7. Not handling spurious interface arrival notification. This can happen and drivers are expected to handle this condition safely.
8. Not implementing a S0 Idle power policy, which is important for devices that are DRIPS constraints or children thereof. See Supporting Idle Power-Down.
9. Not checking WdfDeviceStopIdle return status leads to power reference leak due to WdfDeviceStopIdle/ResumeIdle imbalance and eventually 9F bug check.
10. Not knowing that PrepareHardware/ReleaseHardware can be called more than once due to resource rebalancing. These callbacks should be restricted to initializing hardware resources. See EVT_WDF_DEVICE_PREPARE_HARDWARE.
11. Using PrepareHardware/ReleaseHardware for allocating software resources. Software resource allocation static to the device should be done either in AddDevice or in SelfManagedIoInit if the allocation of resources required interacting with hardware. See EVT_WDF_DEVICE_SELF_MANAGED_IO_INIT.

Coding Guidelines

1. Not using safe string and integer functions. See Using Safe String Functions and Using Safe Integer Functions.
2. Not using typedefs for defining constants.
3. Using globals and static variables. Avoid storing per device context in globals. Globals are meant for sharing

information across multiple instances of devices. As an alternative, consider using WDFDRIVER object context for sharing information across multiple instances of devices.

4. Not using descriptive names for variables.
5. Not being consistent in naming variables - case consistency. Not following the existing style of coding when making updates to existing code. For example, using different variable names for common structures in different functions.
6. Not commenting important design choices - power management, locks, state management, use of workitems, DPCs, timers, global resource usage, resource pre-allocation, complex expressions/conditional statements.
7. Commenting about things that are obvious from the name of the API being called. Making your comment the English language equivalent of the function name (such as writing the comment "Create the Device Object" when calling WdfDeviceCreate).
8. Don't create macros that have a return call. See Functions (C++).
9. No or incomplete Source Code Annotations (SAL). See SAL 2.0 Annotations for Windows Drivers.
10. Using macros instead of inline functions.
11. Using macros for constants in place of constexpr when using C++
12. Compiling your driver with the C compiler, instead of the C++ compiler to ensure you get strong type checking.

Error Handling

1. Not reporting critical driver errors and gracefully marking the device non-functional.
2. Not returning appropriate NT error status that translates to meaningful WIN32 error status. See Using NTSTATUS Values.
3. Not using NTSTATUS macros to check the returned status of system functions.
4. Not asserting on state variables or flags where needed.
5. Checking to see if the pointer is valid before accessing it to work around race conditions.
6. ASSERTING on NULL pointers. If you attempt to use a NULL pointer to access memory Windows will bug check. The parameters of the bug check will provide the necessary information to fix the null pointer. Overtime, when many unneeded ASSERT statements are added to the code, they consume memory, slow the system and make checked build binaries unusable. Note that asserts are not included in the free retail build.
7. ASSERTING on object context pointer. The driver framework guarantees that object will always get allocated with context.

Tracing

1. Not defining WPP custom types and using it in trace calls to get human readable traces messages. See Adding WPP Software Tracing to a Windows Driver.
2. Not using IFR tracing. See Using Inflight Trace Recorder (IFR) in KMDF and UMDF 2 Drivers.
3. Calling out function names in WPP trace calls. WPP already tracks function names and line numbers.
4. Not using ETW events to measure performance and other critical user experience impacting events. See Adding Event Tracing to Kernel-Mode Drivers.
5. Not reporting critical errors in eventlog and gracefully marking the device non-functional.

Verification

1. Not running driver verifier with both standard and advanced settings during development and testing. See Driver Verifier. In the advanced settings, it is recommended to enable all rules, except those rules that are related to low resource simulation. It is preferable to run the low resource simulation tests in isolation to make it easier to debug issues.
2. Not running DevFund test on the driver or the device class the driver is part of with advanced verifier settings enabled. See How to run the DevFund Tests via the command-line.
3. Not verifying that the driver is HVCI compliant. See Evaluate HVCI driver compatibility.
4. Not running AppVerifier on WUDFhost.exe during development and testing of user mode drivers. See Application Verifier.

5. Not checking usage of memory using the !wdfpoolusage debugger extension at runtime to make sure WDF objects are not abandoned. Memory, requests and workitems are common victims of these issues.
6. Not using the !wdfkd debugger extension to inspect the object tree to make sure objects are parented correctly and checking the attributes of major objects such WDFDRIVER, WDFDEVICE, IO.

Windows Kernel-Mode Object Manager

6/25/2019 • 2 minutes to read • Edit Online

The Windows kernel-mode object manager component manages *objects*. Files, devices, synchronization mechanisms, registry keys, and so on, are all represented as objects in kernel mode. Each object has a *header* (containing information about the object such as its name, type, and location), and a *body* (containing data in a format determined by each type of object).

Windows has more than 25 types of objects. A few of the types are:

- Files
- Devices
- Threads
- Processes
- Events
- Mutexes
- Semaphores
- Registry keys
- Jobs
- Sections
- Access tokens
- Symbolic links

The object manager manages the objects in Windows by performing the following major tasks:

- Managing the creation and destruction of objects.
- Keeping an object namespace database for tracking object information.
- Keeping track of resources assigned to each process.
- Tracking access rights for specific objects to provide security.
- Managing the lifetime of an object and determining when an object will be automatically destroyed to recycle resource space.

For more information about objects in Windows, see Device Objects and Device Stacks.

Routines that provide a direct interface to the object manager are usually prefixed with the letters "**Ob**"; for example, **ObGetObjectSecurity**. For a list of object manager routines, see Object Manager Routines.

Note that Windows uses objects as an abstraction for resources. However, Windows is not object-oriented in the classical C++ meaning of the term. Windows is *object-based*. For more information on what object-based means for Windows, see Object-Based.

Windows Kernel-Mode Memory Manager

6/25/2019 • 2 minutes to read • Edit Online

The Windows kernel-mode memory manager component manages physical memory for the operating system. This memory is primarily in the form of random access memory (RAM).

The memory manager manages memory by performing the following major tasks:

- Managing the allocation and deallocation of memory virtually and dynamically.
- Supporting the concepts of memory-mapped files, shared memory, and copy-on-write.

For more detailed information about memory management for drivers, see Memory Management for Windows Drivers.

Routines that provide a direct interface to the memory manager are usually prefixed with the letters "**Mm**"; for example, **MmGetPhysicalAddress**. For a list of memory manager routines, see Memory Manager Routines.

For lists of memory manager routines sorted by functionality, see Memory Allocation and Buffer Management.

Windows Kernel-Mode Process and Thread Manager

10/7/2019 • 3 minutes to read • Edit Online

A *process* is a software program that is currently running in Windows. Every process has an ID, a number that identifies it. A *thread* is an object that identifies which part of the program is running. Each thread has an ID, a number that identifies it.

A process may have more than one thread. The purpose of a thread is to allocate processor time. On a machine with one processor, more than one thread can be allocated, but only one thread can run at a time. Each thread only runs a short time and then the execution is passed on to the next thread, giving the user the illusion that more than one thing is happening at once. On a machine with more than one processor, true multi-threading can take place. If an application has multiple threads, the threads can run simultaneously on different processors.

The Windows kernel-mode process and thread manager handles the execution of all threads in a process. Whether you have one processor or more, great care must be taken in driver programming to make sure that all threads of your process are designed so that no matter what order the threads are handled, your driver will operate properly.

If threads from different processes attempt to use the same resource at the same time, problems can occur. Windows provides several techniques to avoid this problem. The technique of making sure that threads from different processes do not touch the same resource is called *synchronization*. For more information about synchronization, see Synchronization Techniques.

Routines that provide a direct interface to the process and thread manager are usually prefixed with the letters "**Ps**"; for example, **PsCreateSystemThread**. For a list of kernel DDIs, see Windows kernel.

This set of guidelines applies to these callback routines:

PCREATE_PROCESS_NOTIFY_ROUTINE

PCREATE_PROCESS_NOTIFY_ROUTINE_EX

PCREATE_THREAD_NOTIFY_ROUTINE

PLOAD_IMAGE_NOTIFY_ROUTINE

- Keep notify routines short and simple.
- Do not make calls into a user mode service to validate the process, thread, or image.
- Do not make registry calls.
- Do not make blocking and/or Interprocess Communication (IPC) function calls.
- Do not synchronize with other threads because it can lead to reentrancy deadlocks.
- Use System Worker Threads to queue work especially work involving:
  - Slow API's or API's that call into other process.
  - Any blocking behavior which could interrupt threads in core services.
- Be considerate of best practices for kernel mode stack usage. For examples, see How do I keep my driver from running out of kernel-mode stack? and Key Driver Concepts and Tips.

Subsystem Processes

Starting in Windows 10, the Windows Subsystem for Linux (WSL) enables a user to run native Linux ELF64 binaries on Windows, alongside other Windows applications. For information about WSL architecture and the user-mode and kernel-mode components that are required to run the binaries, see the posts on the Windows Subsystem for Linux blog.

One of the components is a *subsystem process* that hosts the unmodified user-mode Linux binary, such as /bin/bash. Subsystem processes do not contain data structures associated with Win32 processes, such as Process Environment Block (PEB) and Thread Environment Block (TEB). For a subsystem process, system calls and user mode exceptions are dispatched to a paired driver.

Here are the changes to the Process and Thread Manager Routines in order to support subsystem processes:

- The WSL type is indicated by the **SubsystemInformationTypeWSL** value in the **SUBSYSTEM_INFORMATION_TYPE** enumeration. Drivers can call **NtQueryInformationProcess** and **NtQueryInformationThread** to determine the underlying subsystem. Those calls return **SubsystemInformationTypeWSL** for WSL.
- Other kernel mode drivers can get notified about subsystem process creation/deletion by registering their callback routine through the **PsSetCreateProcessNotifyRoutineEx2** call. To get notifications about thread creation/deletion, drivers can call **PsSetCreateThreadNotifyRoutineEx**, and specify **PsCreateThreadNotifySubsystems** as the type of notification.
- The **PS_CREATE_NOTIFY_INFO** structure has been extended to include a **IsSubsystemProcess** member that indicates a subsystem other than Win32. Other members such as **FileObject**, **ImageFileName**, **CommandLine** indicate additional information about the subsystem process. For information about the behavior of those members, see **SUBSYSTEM_INFORMATION_TYPE**.

Windows Kernel-Mode I/O Manager

6/25/2019 • 2 minutes to read • Edit Online

A computer consists of various devices that provide input and output (I/O) to and from the outside world. Typical devices are keyboards, mice, audio controllers, video controllers, disk drives, networking ports, and so on. Device drivers provide the software connection between the devices and the operating system. For this reason, I/O is very important to the device driver writer.

The Windows kernel-mode I/O manager manages the communication between applications and the interfaces provided by device drivers. Because devices operate at speeds that may not match the operating system, the communication between the operating system and device drivers is primarily done through I/O request packets (IRPs). These packets are similar to network packets or Windows message packets. They are passed from operating system to specific drivers and from one driver to another.

The Windows I/O system provides a layered driver model called stacks. Typically IRPs go from one driver to another in the same stack to facilitate communication. For example, a joystick driver would need to communicate to a USB hub, which in turn would need to communicate to a USB host controller, which would then need to communicate through a PCI bus to the rest of the computer hardware. The stack consists of joystick driver, USB hub, USB host controller, and the PCI bus. This communication is coordinated by having each driver in the stack send and receive IRPs.

It cannot be stressed enough that your driver must send and receive IRPs on a timely basis for the whole stack to operate efficiently. If your driver is part of a stack and does not properly receive, handle, and pass on the information, your driver may cause system crashes.

For more information about IRPs, see Handling IRPs.

For more information about driver stacks, see Device Objects and Device Stacks.

For programming techiques related to I/O management, see I/O Manager Programming Techniques.

Routines that provide a direct interface to the I/O manager are usually prefixed with the letters "**Io**"; for example, **IoCreateDevice**. For a list of I/O manager routines, see I/O Manager Routines.

For lists of routines that relate to IRPS, see IRPs.

The I/O manager has two subcomponents: the Plug and Play manager and power manager. They manage the I/O functionality for the technologies of Plug and Play and power management. For more information about Plug and Play management, see Windows Kernel-Mode Plug and Play Manager and for more information about power management, see Windows Kernel-Mode Power Manager.

Windows Kernel-Mode Plug and Play Manager

6/25/2019 • 2 minutes to read • Edit Online

Plug and Play (PnP) is a combination of hardware technology and software techniques that enables a PC to recognize when a device is added to the system. With PnP, the system configuration can change with little or no input from the user. For example, when a USB thumb drive is plugged in, Windows can detect the thumb drive and add it to the file system automatically. However, to do this, the hardware must follow certain requirements and so must the driver.

For more information about PnP for drivers, see Plug and Play.

The PnP manager is actually a subsystem of the I/O manager. For more information about the I/O manager, see Windows Kernel-Mode I/O Manager.

For lists of PnP routines, see Plug and Play Routines.

Note that there are no routines that provide a direct interface to the PnP manager; that is, there are no "**Pp**" routines.

The Windows Driver Frameworks (WDF) provide a set of libraries to make PnP management much easier. For more information about WDF, see Kernel-Mode Driver Framework Overview.

Windows Kernel-Mode Power Manager

6/25/2019 • 2 minutes to read • Edit Online

Windows uses power management technology to reduce power consumption for PCs in general and for battery-powered laptops in particular. For example, a Windows computer can be put in a sleep or hibernation state. A complex power management system for computer devices has evolved so that when the computer begins to shut down or go to lower power consumption, the attached devices can also be powered down in a proper manner so that no data is lost. But these devices need a warning that the power status in changing and they may also need to be part of a communications loop that tells the controlling device to wait until they can shut down properly.

The Windows kernel-mode power manager manages the orderly change in power status for all devices that support power state changes. This is often done through a complex stack of devices controlling other devices. Each controlling device is called a *node* and must have a driver that can handle the communication of power state changes up and down through a device stack.

If you are writing a driver that can be affected by power-state changes, you must be able to process the following types of information in your driver code:

- System activity level.
- System battery level.
- Current requests to shut down, sleep, or hibernate.
- User actions such as pressing a power button.
- Control panel settings, such as automatically shutting down at 10 percent battery power.

The power manager handles these requests using IRPs. For more information about IRPs, see Handling IRPs.

The power manager works in combination with policy management to handle power management and coordinate power events, and then generates power management IRPs. The power manager collects requests to change the power state, decides which order the devices must have their power state changed, and then send the appropriate IRPs to tell the appropriate drivers to make the changes (which in turn may tell subdevices to make the change as well). The policy manager monitors activity in the system and integrates user status, application status, and device driver status into power policy.

For more detailed information about power management, see Power Management for Windows Drivers.

The power manager is considered a subcomponent of the I/O manager. For more information, see Windows I/O Manager.

Routines that provide a direct interface to the power manager are usually prefixed with "**Po**"; for example, **PoSetPowerState**. For a list of power manager routines, see Power Manager Routines.

The Windows Driver Frameworks (WDF) provides a set of libraries to make power management much easier. For more information about WDF, see Kernel-Mode Driver Framework Overview.

Windows Kernel-Mode Configuration Manager

6/25/2019 • 2 minutes to read • Edit Online

In the earlier days of Microsoft Windows, applications and the operating system stored configuration values in "INI" (initialization) files. This provided a simple way to store state values that could be preserved from one Windows session to the next. However, as the Windows environment became more complex, a new system of storing persistent information about the operating system and applications was needed. The Windows Registry was created to store data about hardware and software.

The Windows kernel-mode configuration manager manages the registry. If your driver needs to know about changes in the registry, it can use the routines of the configuration manager to do so by registering callbacks on specific registry data. Then, when the data in the registry changes, the callback is triggered and you can run code to process the callback information in your driver.

Routines that provide a direct interface to the configuration manager are prefixed with the letters "**Cm**"; for example, **CmRegisterCallback**. For a list of configuration manager routines, see Configuration Manager Routines.

In addition to directly calling the configuration manager, there are other ways you will want to work with the registry in your driver. For more information about using the registry in a driver, see Using the Registry in a Driver and Registry Keys for Drivers.

Windows Kernel-Mode Kernel Transaction Manager

12/21/2018 • 2 minutes to read • Edit Online

When you are dealing with multiple reads and writes on one or more data stores, and the operations must all atomically succeed or fail to preserve the integrity of the data, you might want to group the operations together as a single transaction. If all of the operations within the transaction succeed, the transaction can be committed so that all the changes persist as an atomic unit. If a failure occurs, the transaction can be rolled back so that the data stores are restored to their original state.

The kernel transaction manager (KTM) is the Windows kernel-mode component that implements transaction processing in kernel mode. KTM allows kernel mode components, such as drivers, to perform transactions. In addition, KTM is the platform on which user-mode Transactional NTFS (TxF) is based.

For information about how to use KTM in kernel-mode components, see Kernel Transaction Manager.

Windows Kernel-Mode Security Reference Monitor

6/25/2019 • 2 minutes to read • Edit Online

An increasingly important aspect of operating systems is security. Before an action can take place, the operating system must be sure that the action is not a violation of system policy. For example, a device may or may not be accessible to all requests. When creating a driver, you may want to allow some requests to succeed or fail, depending on the permission of the entity making the request.

Windows uses an access control list (ACL) to determine which objects have what security. The Windows kernel-mode security reference monitor provides routines for your driver to work with access control. For more information about the ACL, see Access Control List.

Routines that provide a direct interface to the security reference monitor are prefixed with the letters "**Se**"; for example, **SeAccessCheck**. For a list of security reference monitor routines, see Security Reference Monitor Routines.

Windows Kernel-Mode Kernel Library

6/25/2019 • 2 minutes to read • Edit Online

The *kernel* of an operating system implements the core functionality that everything else in the operating system depends upon. The Microsoft Windows kernel provides basic low-level operations such as scheduling threads or routing hardware interrupts. It is the heart of the operating system and all tasks it performs must be fast and simple.

Routines that provide a direct interface to the kernel library are usually prefixed with "**Ke**", for example, **KeGetCurrentThread**. For a list of kernel library routines, see Kernel Library Support Routines.

Note The term *microkernel* does not apply to the current kernel used in the Windows operating system.

Windows Kernel-Mode Executive Support Library

6/25/2019 • 2 minutes to read • Edit Online

The Windows operating system uses the term *executive layer* to refer to kernel-mode components that provide a variety of services to device drivers, including:

- Object management
- Memory management
- Process and thread management
- Input/output management
- Configuration management

Each of the above managers provides direct interfaces to their individual technologies, as do several libraries. However, routines that are grouped together as a generic interface to the Executive Library are usually prefixed with "**Ex**", for example, **ExGetCurrentResourceThread**. For a list of executive library routines, see Executive Library Support Routines.

Note that the executive layer components are part of Ntoskrnl.exe, but that drivers and the HAL are not part of the executive layer.

Windows Kernel-Mode Run-Time Library

6/25/2019 • 2 minutes to read • Edit Online

Windows provides a set of common utility routines needed by various kernel-mode components. For example, **RtlCheckRegistryKey** is used to see if a given key is in the registry.

Most of the run-time library (RTL) routines are prefixed with the letters "**Rtl**"; for a list of the run-time library routines for the kernel, see Run-Time Library (RTL) Routines.

There is also a different kernel-mode library specifically designed for safe string handling. For more information about the safe string library, see Windows Kernel-Mode Safe String Library. Note that safe string library routines are also usually prefixed by "**Rtl**" but are not part of the run-time library (RTL).

Windows Kernel-Mode Safe String Library

12/5/2018 • 2 minutes to read • Edit Online

One of the major problems in software security is related to the vulnerability of working with strings. To provide greater security, Windows provides a safe string library.

Safe string library routines are prefixed with the letters "**Rtl**"; for a list of all safe string library routines for the kernel, see Safe String Functions for Unicode and ANSI Characters and Safe String Functions for UNICODE_STRING Structures.

For more information about using safe strings, see Using Safe String Functions.

Note that there is also a separate run-time library for general C programming in the kernel that has string functionality as well. For more information about the run-time library (RTL), see Windows Kernel-Mode Run-Time Library. Note that even though both libraries are prefixed with "**Rtl**" they are not the same library.

Windows Kernel-Mode DMA Library

6/25/2019 • 2 minutes to read • Edit Online

To enhance performance, a device may need direct access to memory in a way that bypasses the central processing unit (CPU). This technology is called direct memory access (DMA). Windows provides a DMA library for device driver developers.

For more information about DMA for drivers, see DMA.

For a listing of DMA routines, see Direct Memory Access (DMA) Library Routines.

Note that DMA is a technology for communicating directly between device and memory and is not the same as Device Memory Access, which is a set of macros provided to read and write to I/O ports and CPU registers.

Windows Kernel-Mode HAL Library

6/25/2019 • 2 minutes to read • Edit Online

Windows runs on many different configurations of the personal computer. Each configuration requires a layer of software that interacts between the hardware and the rest of the operating system. Because this layer abstracts (hides) the low-level hardware details from drivers and the operating system, it is called the hardware abstraction layer (HAL).

Developers are not encouraged to write their own HAL. If you need hardware access, the HAL library provides routines that can be used for that purpose. Routines that interface with the HAL directly are prefixed with the letters "**Hal**"; for a list of HAL routines, see Hardware Abstraction Layer (HAL) Library Routines.

Windows Kernel-Mode CLFS Library

6/25/2019 • 2 minutes to read • Edit Online

Windows provides a transactional logging system for system files. This system is called the Common Log File System (CLFS). For more information about CLFS, see Common Log File System.

Routines that provide a direct interface for CLFS are prefixed with the letters "**Clfs**"; for a list of CLFS library routines, see Common Log File System (CLFS) Library Routines. CLFS also provides a list of routines that you can implement to manage a CLFS; for more information on CLFS management, see CLFS Management Library Routines.

CLFS is a technology that is related to transacted file systems; for more information about transactions, see Windows Kernel-Mode Transaction Manager.

Windows Kernel-Mode WMI Library

6/25/2019 • 2 minutes to read • Edit Online

Windows provides a general mechanism for managing components. This system is called Windows Management Instrumentation (WMI). To satisify Windows Driver Model (WDM) requirements, you should implement WMI for your driver so that your driver can be managed by the system.

For more information on WMI, see Windows Management Instrumentation.

Routines that provide a direct interface to the WMI library are prefixed with the letters "**Wmi**"; for a list of WMI routines, see Windows Management Instrumentation (WMI) Library Routines.

For a list of WMI callbacks, see WMI Library Callback Routines.

Communication with WMI is done with IRPs. For a list of routines that your driver can use to receive IRPs, see WMI IRP Processing Routines. For a list of routines that your driver can use to send WMI IRPs, see WMI IRP Sending Routines. For a list of IRPs that are used with WMI, see WMI Minor IRPs.

Writing WDM Drivers

6/25/2019 • 2 minutes to read • Edit Online

This section discusses the Microsoft Windows Driver Model (WDM) architecture. This architecture started in Windows 2000 as an enhancement to previous Windows NT device drivers.

Note Drivers for versions of Windows NT-based operating systems before Windows 2000 are not supported, and you should update these drivers. The WDM architecture does not support drivers for non-Windows NT-based operating systems (such as Windows 98), and you should rewrite such drivers.

This section is divided into three parts:

- Windows Driver Model describes the Windows Driver Model (WDM), including types of WDM drivers, device configuration, and WDM versioning.
- Device Objects and Device Stacks describes device objects and device stacks. The section includes information about physical device objects (PDOs), functional device objects (FDOs), and filter device objects (filter DOs). Drivers are often built from a set of device objects that work together. This set of device objects is called a *stack*. Stacks can help you understand the flow of information to and from a driver and how different parts of the driver communicate internally.
- Kernel-Mode Driver Components describes which routines you must implement to have a functional driver and which routines are optional.

  A *device driver* is a set of software code that must integrate into the operating system. To complete this integration, you must write a set of handler routines in your driver that process calls from the operating system. These routines can be simple function calls, but many of them implement the processing of *I/O request packets* (IRPs), which facilitate communication between drivers and the operating system.

Note WDM drivers can also use the Windows Driver Frameworks (WDF) library to make some parts of a device driver easier to write. Specifically, kernel-mode drivers can use the Kernel-Mode Driver Framework (KMDF), which is part of WDF. For more information about KMDF for kernel-mode drivers, see Kernel-Mode Driver Framework Overview. Note that KMDF does not replace WDM. You must still understand many parts of WDM to write a KMDF driver.

Introduction to WDM

6/25/2019 • 2 minutes to read • Edit Online

> **NOTE**
> This section contains guidance on WDM drivers, which is no longer the recommended driver model. For guidance on choosing a driver model, see Choosing a driver model.

To allow driver developers to write device drivers that are source-code compatible across all Microsoft Windows operating systems, the *Windows Driver Model* (WDM) was introduced. Kernel-mode drivers that follow WDM rules are called *WDM drivers*.

All WDM drivers must do the following:

- Include Wdm.h, not Ntddk.h. (Note that Wdm.h is a subset of Ntddk.h.)
- Be designed as a bus driver, a function driver, or a filter driver, as described inTypes of WDM Drivers.
- Create device objects as described in WDM Device Objects and Device Stacks.
- Support Plug and Play (PnP).
- Support power management.
- Support Windows Management Instrumentation (WMI).

Should You Write a WDM Driver?

If you are writing a new driver, consider using the Kernel-Mode Driver Framework (KMDF). KMDF provides interfaces that are simpler to use than WDM interfaces.

Do not write a WDM driver if the driver will be inserted into a stack of non-WDM drivers. Please read the documentation for device type-specific Microsoft-supplied drivers to determine how new drivers must interface with Microsoft-supplied drivers. For more device type-specific information, see Device and Driver Technologies.)

Types of WDM Drivers

10/7/2019 • 2 minutes to read • Edit Online

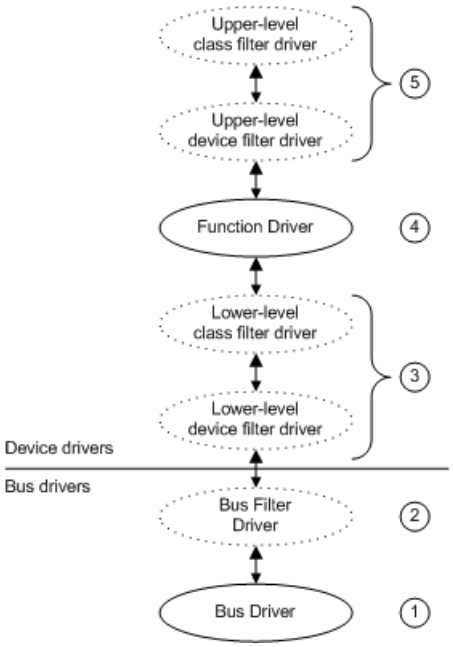
There are three kinds of WDM drivers: bus drivers, function drivers, and filter drivers.

- A bus driver drives an individual I/O bus device and provides per-slot functionality that is device-independent. Bus drivers also detect and report child devices that are connected to the bus.
- A function driver drives an individual device.
- A filter driver filters I/O requests for a device, a class of devices, or a bus.

In this context, a *bus* is any device to which other physical, logical, or virtual devices are attached; a bus includes traditional buses such as SCSI and PCI, as well as parallel ports, serial ports, and i8042 ports.

It is important for driver developers to understand the different kinds of WDM drivers and to know which kind of driver they are writing. For example, whether a driver handles each Plug and Play IRP and how to handle such IRPs depends on what kind of driver is being written (bus driver, function driver, or filter driver).

The following figure shows the relationship between the bus driver, function driver, and filter drivers for a device.



Each device typically has a bus driver for the parent I/O bus, a function driver for the device, and zero or more filter drivers for the device. A driver design that requires many filter drivers does not yield optimal performance.

The drivers in the previous figure are the following:

1. A *bus driver* services a bus controller, adapter, or bridge. Bus drivers are required drivers; there is one bus driver for each type of bus on a machine. Microsoft provides bus drivers for most common buses. IHVs and OEMs can provide other bus drivers.

2. A *bus filter driver* typically adds value to a bus and is supplied by Microsoft or a system OEM. There can be any number of bus filter drivers for a bus.

3. *Lower-level filter drivers* typically modify the behavior of device hardware. They are optional and are typically supplied by IHVs. There can be any number of lower-level filter drivers for a device.

4. A *function driver* is the main driver for a device. A function driver is typically written by the device vendor and is required (unless the device is being used in *raw mode*).

5. *Upper-level filter drivers* typically provide added-value features for a device. They are optional and are typically provided by IHVs.

The following topics describe the three general types of WDM drivers—bus drivers, function drivers, filter drivers—in detail. Also included is an example of WDM driver layering that uses sample USB drivers.

In this section

- Bus Drivers
- Function Drivers
- Filter Drivers
- WDM Driver Layers: An Example

Bus Drivers

1/11/2019 • 2 minutes to read • Edit Online

A *bus driver* services a bus controller, adapter, or bridge (see the Possible Driver Layers figure). Microsoft provides bus drivers for most common buses, such as PCI, PnpISA, SCSI, and USB. Other bus drivers can be provided by IHVs or OEMs. Bus drivers are required drivers; there is one bus driver for each type of bus on a machine. A bus driver can service more than one bus if there is more than one bus of the same type on the machine.

The primary responsibilities of a bus driver are to:

- Enumerate the devices on its bus.
- Respond to Plug and Play IRPs and power management IRPs.
- Multiplex access to the bus (for some buses).
- Generically administer the devices on its bus.

Bus drivers are essentially function drivers that also enumerate children.

During enumeration, a bus driver identifies the devices on its bus and creates device objects for them. (For information about device objects, see Device Objects and Device Stacks.) The method a bus driver uses to identify connected devices depends on the particular bus.

A bus driver performs certain operations on behalf of the devices on its bus, including accessing device registers to physically change the power state of a device. For example, when the device goes to sleep, the bus driver sets device registers to put the device in the proper device power state.

Note, however, that a bus driver does not handle read and write requests for the child devices that are connect to its bus. Read and write requests to a child device are handled by the child device's function driver does the parent bus driver handle reads and writes for the device.

Because a bus driver acts as the function driver for its controller, adapter, or bridge, it also manages device power policy for these components.

A bus driver can be implemented as a driver/minidriver pair, the way a SCSI port/miniport driver pair drives a SCSI host bus adapter (HBA). In such driver pairs, the minidriver is linked to the second driver, which is a DLL.

Function Drivers

1/11/2019 • 2 minutes to read • Edit Online

A *function driver* is the main driver for a device (see the Possible Driver Layers). The PnP manager loads at most one function driver for a device. A function driver can service one or more devices.

A function driver provides the operational interface for its device. Typically the function driver handles reads and writes to the device and manages device power policy.

The function driver for a device can be implemented as a driver/minidriver pair, such as a port/miniport driver pair or a class/miniclass driver pair. In such driver pairs, the minidriver is linked to the second driver, which is a DLL.

If a device is being driven in raw mode, it has no function driver and no upper or lower-level filter drivers. All raw-mode I/O is done by the bus driver and optional bus filter drivers.

Filter Drivers

10/7/2019 • 2 minutes to read • Edit Online

Filter drivers are optional drivers that add value to or modify the behavior of a device. A filter driver can service one or more devices.

Bus Filter Drivers

Bus filter drivers typically add value to a bus and are supplied by Microsoft or a system OEM (see the Possible Driver Layers figure). Bus filter drivers are optional. There can be any number of bus filter drivers for a bus.

A bus filter driver could, for example, implement proprietary enhancements to standard bus hardware.

For devices described by an ACPI BIOS, the power manager inserts a Microsoft-supplied *ACPI filter* (bus filter driver) above the bus driver for each such device. The ACPI filter carries out device power policy and powers on and off devices. The ACPI filter is transparent to other drivers and is not present on non-ACPI machines.

Lower-Level Filter Drivers

Lower-level filter drivers typically modify the behavior of device hardware (see the Possible Driver Layers figure). They are typically supplied by IHVs and are optional. There can be any number of lower-level filter drivers for a device.

A lower-level *device* filter driver monitors and/or modifies I/O requests to a particular device. Typically, such filters redefine hardware behavior to match expected specifications.

A lower-level *class* filter driver monitors and/or modifies I/O requests for a class of devices. For example, a lower-level class filter driver for mouse devices could provide acceleration, performing a nonlinear conversion of mouse movement data.

Upper-Level Filter Drivers

Upper-level filter drivers typically provide added-value features for a device (see the Possible Driver Layers figure). Such drivers are usually provided by IHVs and are optional. There can be any number of upper-level filter drivers for a device.

An upper-level *device* filter driver adds value for a particular device. For example, an upper-level device filter driver for a keyboard could enforce additional security checks.

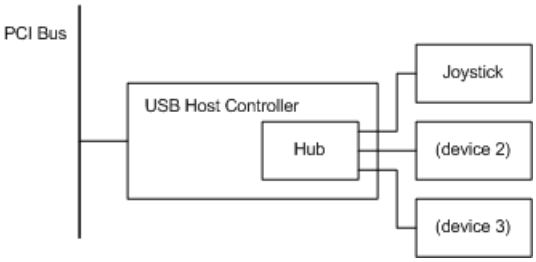
An upper-level *class* filter driver adds value for all devices of a particular class.

WDM Driver Layers: An Example

12/5/2018 • 2 minutes to read • Edit Online

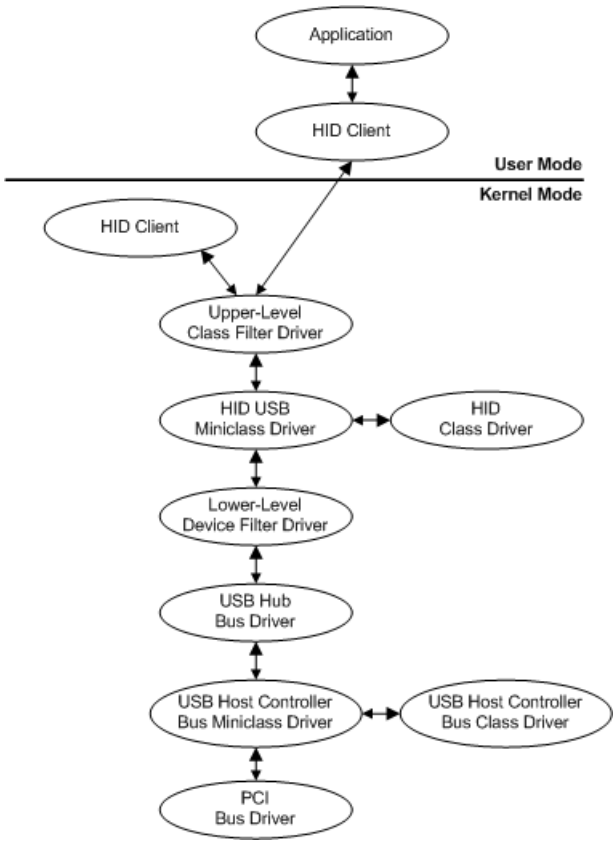
This section describes a possible set of WDM drivers for USB hardware to illustrate WDM driver layers.

The following figure shows a sample PnP hardware configuration for a USB joystick.



In this figure, the USB joystick plugs into a port on a USB hub. The USB hub in this example resides on the USB Host Controller board and is plugged into the single port on the USB host controller board. The USB host controller plugs into a PCI bus. From a PnP perspective, the USB hub, the USB host controller, and the PCI bus are all bus devices because they each provide ports. The joystick is not a bus device.

The following figure shows a sample set of drivers that might be loaded for the USB joystick hardware in the previous figure.



Starting at the bottom of the previous figure, the drivers in the sample stack include:

- A PCI driver that drives the PCI bus. This is a PnP bus driver. The PCI bus driver is provided with the system by Microsoft.
- The bus driver for the USB host controller is implemented as a class/miniclass driver pair. The USB host controller class and miniclass drivers are provided with the system by Microsoft.

- The USB hub bus driver that drives the USB hub. The USB hub driver is provided with the system by Microsoft.
- Three drivers for the joystick device; one of them is a class/miniclass pair.

  The function driver, the main driver for the joystick device, is the HID class driver/HID USB miniclass driver pair. (HID represents "Human Interface Device".) The HID USB miniclass driver supports the USB-specific semantics of HID devices, relying on the HID class driver DLL for general HID support.

  A function driver can be specific to a particular device, or, as in the case of HID, a function driver can service a group of devices. In this example, the HID class driver/HID USB miniclass driver pair services any HID-compliant device in the system on a USB bus. A HID class driver/HID 1394 miniclass driver pair would service any HID-compliant device on a 1394 bus.

  A function driver can be written by the device vendor or by Microsoft. In this example, the function driver (the HID class/HID USB miniclass driver pair) is written by Microsoft.

  There are two filter drivers for the joystick device in this example: an upper-level class filter that adds a macro button feature and a lower-level device filter that enables the joystick to emulate a mouse device.

  The upper-level filter is written by someone who needs to filter the joystick I/O and the lower-level filter driver is written by the joystick vendor.
- The kernel-mode and user-mode HID clients and the application are not drivers but are shown for completeness.

Device Configurations and Layered Drivers

12/5/2018 • 2 minutes to read • Edit Online

For the most common kinds of devices, the Windows Driver Kit (WDK) supplies a sample set of fully functional system drivers. Individual sample drivers can be used as models when developing new drivers for similar kinds of devices. However, the system's drivers had an additional design requirement: to make it easy to develop new device drivers. Consequently, many of the system's drivers have a layered architecture so that certain drivers can be reused to support new drivers for similar devices.

In most cases, the WDK-supplied reusable drivers are WDM drivers that support PnP and handle hardware-independent operations for a system-supplied device-specific lowest-level (PnP bus) driver. In some cases, such as the parallel port and SCSI port drivers, these reusable drivers provide support for higher-level, device-type-specific class drivers. Note that none of the system's reusable drivers precludes the development of new intermediate drivers to be added to a chain of existing drivers.

Where a new (or replacement) driver fits in the chain of drivers for a device depends partly on the hardware configuration of devices in a given Windows platform, and partly on how much support a new driver can get from existing system drivers.

In this section

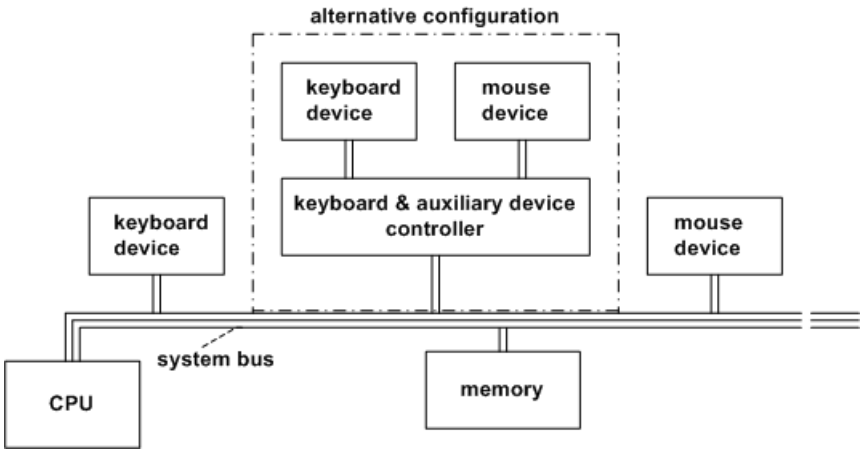
Sample Device and Driver Configuration

1/11/2019 • 2 minutes to read • Edit Online

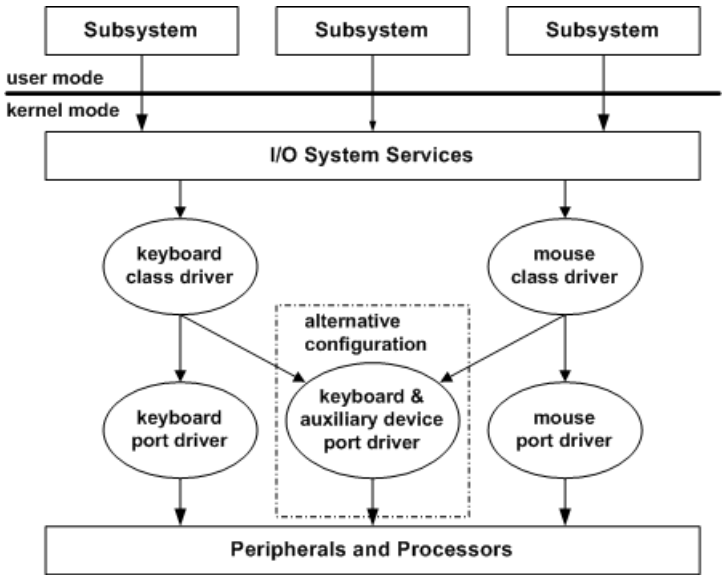
This section illustrates the relationship between the hardware and driver configurations, using the keyboard and mouse devices as an example. Configurations differ for other devices. For complete information about any device configuration, see the device-specific documentation in the Windows Driver Kit (WDK).

The following figure shows two possible hardware configurations for the keyboard and mouse devices:

- Each connected directly somewhere on the system bus
- Both connected through a keyboard and auxiliary device controller



The following figure illustrates the corresponding layered drivers for I/O operations on the devices shown in the previous figure.



Note that drivers of keyboard and mouse devices, whatever the hardware configuration, can use the system's keyboard class and mouse class drivers to handle hardware-independent operations. These are called *class drivers* because each supplies system-required but hardware-independent support for a particular class of device.

A corresponding *port driver* implements the device-specific support to carry out required I/O operations on each physical device. The system's (i8042) keyboard and auxiliary device port driver for x86-based platforms manages device-specific operations for both mouse and keyboard. In a hardware configuration where each device is separately connected, as shown in the figure illustrating the keyboard and mouse hardware configurations, each

system class driver can be layered over separate device-specific port drivers, or a single driver for each device could be implemented as a separate, monolithic (lowest-level) driver.

A new intermediate driver, such as a PnP filter driver, could be added to the configuration in the figure illustrating the keyboard and mouse driver layers. For example, a filter driver added above the keyboard class driver might filter keyboard input in a platform-specific manner before passing it through the I/O services to the subsystem that requested it. Such a filter driver must recognize the same IRPs and IOCTLs as the keyboard class driver.

Points to Consider When Adding Drivers

12/5/2018 • 2 minutes to read • Edit Online

Keep the following points in mind when designing a kernel-mode driver:

- The system-supplied SCSI and video port drivers cannot be replaced.
- A replacement lowest-level driver must implement the same functionality as the driver it replaces. For example, a replacement keyboard or mouse port driver must use the system-defined interface between itself and a system-supplied class driver that it reuses, and vice versa.
- A new intermediate driver, inserted between any pair of system-supplied drivers, must interoperate with those drivers so that the functionality of the upper and lower drivers is not reduced.

Determining the WDM Version

6/25/2019 • 2 minutes to read • Edit Online

A cross-system WDM driver should use the **IoIsWdmVersionAvailable** routine to determine which version of WDM is supported by the system on which it is running. The reference page for **IoIsWdmVersionAvailable** provides a list of WDM version numbers.

For information about differences in WDM that drivers should handle, see Differences in WDM Versions.

Differences in WDM Versions

6/25/2019 • 2 minutes to read • Edit Online

The simplest way to ensure cross-system compatibility is to write a driver that uses only features that are supported by the lowest-numbered version of WDM. However, this is not always possible. Sometimes, drivers require additional code to take advantage of the features that are available in later versions of WDM, or to compensate for differences between Windows operating systems.

WDM Differences in Driver Support Routines

The Windows Driver Kit (WDK) reference page for each driver support routine indicates if the routine is restricted to specific versions of WDM, or if its behavior is different on different operating system versions. Before using any driver support routine in a cross-system driver, be sure to understand any version-specific restrictions or behaviors.

WDM Differences in Plug and Play

The following Plug and Play I/O request packet (IRP) is supported only in Windows 2000 and later versions of the NT-based operating system (WDM version 1.10 and later):

IRP_MN_SURPRISE_REMOVAL

In addition, the following IRPs work differently on Windows 98/Me from how they work on the NT-based operating system:

IRP_MN_STOP_DEVICE and **IRP_MN_REMOVE_DEVICE**

IRP_MN_QUERY_REMOVE_DEVICE

WDM Differences in Power Management

The following power management functions and I/O requests differ in operation between the Windows 98/Me operating system and the NT-based operating system:

PoSetPowerState

PoRequestPowerIrp

PoRegisterDeviceForIdleDetection

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

When completing power IRPs, drivers on Windows 98/Me must complete power IRPs at IRQL = PASSIVE_LEVEL, while drivers on the NT-based operating system can complete such IRPs at IRQL = PASSIVE_LEVEL or IRQL = DISPATCH_LEVEL.

The DO_POWER_PAGABLE flag in the **DEVICE_OBJECT** structure is used differently on the Windows 98/Me operating system than on the NT-based operating system.

WDM Differences in Kernel-Mode Driver Operation

Kernel-mode WDM drivers for Windows 98/Me must follow certain guidelines for using floating-point operations, MMX, 3DNOW!, or Intel's SSE extensions. For more information, see Using Floating Point or MMX in a WDM Driver.

Windows 98/Me provides a fixed number of worker threads that might not be adequate for some drivers.

Kernel-Mode Driver Components

12/5/2018 • 2 minutes to read • Edit Online

This section introduces the standard routines contained in kernel-mode drivers. Some of these *standard driver routines* are required; others are optional. The section also introduces *driver objects*, which contain pointers to each driver's standard routines.

Some drivers interact with a system-supplied port driver or class driver that defines much of the driver's required functionality. For example, a SCSI miniport driver primarily interacts with the SCSI port driver. For such drivers, see the class-specific documentation for details of required and optional driver support.

This section includes:

Introduction to Standard Driver Routines

6/25/2019 • 2 minutes to read • Edit Online

Each kernel-mode driver is constructed around a set of system-defined, standard driver routines. Kernel-mode drivers process *I/O request packets* (IRPs) within these standard routines by calling system-supplied driver support routines.

All drivers, regardless of their level in a chain of attached drivers, must have a basic set of standard routines in order to process IRPs. Whether a driver must implement additional standard routines depends on whether the driver controls a physical device or is layered over a physical device driver, as well as on the nature of the underlying physical device. Lowest-level drivers that control physical devices have more required routines than higher-level drivers, which typically pass IRPs to a lower driver for processing.

Standard driver routines can be divided into two groups: those that each kernel-mode driver must have, and those that are optional, depending on the driver type and location in the *device stack*.

This section describes required standard routines. Other sections describe the optional routines.

Following are two tables. The first table lists required standard routines. The second lists most of the optional routines.

REQUIRED STANDARD DRIVER ROUTINES	PURPOSE	WHERE DESCRIBED
DriverEntry	Initializes the driver and its driver object.	Writing a DriverEntry Routine
AddDevice	Initializes devices and creates device objects.	Writing an AddDevice Routine
Dispatch Routines	Receive and process IRPs.	Writing Dispatch Routines
Unload	Release system resources acquired by the driver.	Writing an Unload Routine

OPTIONAL STANDARD DRIVER ROUTINES	PURPOSE	WHERE DESCRIBED
Reinitialize	Completes driver initialization if **DriverEntry** cannot.	Writing a Reinitialize Routine
StartIo	Starts an I/O operation on a physical device.	Writing a StartIo Routine
Interrupt Service Routine	Saves the state of a device when it interrupts.	Writing an ISR

OPTIONAL STANDARD DRIVER ROUTINES	PURPOSE	WHERE DESCRIBED
Deferred Procedure Calls	Completes the processing of a device interrupt after an ISR saves the device state.	DPC Objects and DPCs
SynchCritSection	Synchronizes access to driver data.	Using Critical Sections
AdapterControl	Initiates DMA operations.	Adapter Objects and DMA
IoCompletion	Completes a driver's processing of an IRP.	Completing IRPs
Cancel	Cancels a driver's processing of an IRP.	Canceling IRPs
CustomTimerDpc, *IoTimer*	Timing and synchronizing events.	Synchronization Techniques

The current IRP and target device object are input parameters to many standard routines. Every driver processes each IRP in stages through its set of standard routines.

By convention, the system-supplied drivers prepend an identifying, driver-specific or device-specific prefix to the name of every standard routine except **DriverEntry**. As an example, this documentation uses "DD", as shown in the driver object illustration. Following this convention makes it easier to debug and maintain drivers.

Standard Driver Routine Requirements

6/25/2019 • 2 minutes to read • Edit Online

Keep the following points in mind when designing a kernel-mode driver:

- Each driver must have a **DriverEntry** routine, which initializes driver-wide data structures and resources. The I/O manager calls the **DriverEntry** routine when it loads the driver.
- Every driver must have at least one dispatch routine that receives and processes I/O request packets (IRPs). Each driver must place a dispatch routine's entry point in its **DRIVER_OBJECT** structure, for each IRP major function code that the driver can receive. A driver can have a separate dispatch routine for each IRP major function code, or it can have one or more dispatch routines that handle several function codes.
- Every WDM driver must have an *Unload* routine. The driver must place the *Unload* routine's entry point in the driver's driver object. The responsibilities of a PnP driver's Unload routine are minimal, but a non-PnP driver's unload routine is responsible for releasing any system resources that the driver is using.
- Every WDM driver must have an *AddDevice* of the driver object. An *AddDevice* routine is responsible for creating and initializing device objects for each PnP device the driver controls.
- A driver can have a *StartIo* routine, which the I/O manager calls to start I/O operations for IRPs the driver has queued to a system-supplied IRP queue. Any driver that does not have a *StartIo* routine must either set up and manage internal queues for the IRPs it receives, or it must complete every IRP within its dispatch routines. Higher-level drivers might not have a *StartIo* routine, if they simply pass IRPs to lower-level drivers directly from their dispatch routines.
- Certain miniport drivers are exceptions to the preceding requirements. See the device-type-specific documentation in the Windows Driver Kit (WDK) for information about the requirements for miniport drivers.
- Whether a driver has any other kind of standard routine depends on its functionality and on how that driver fits into the system (for example, whether it interoperates with system-supplied drivers). See the device-type specific documentation in the WDK for details.

Introduction to Driver Objects

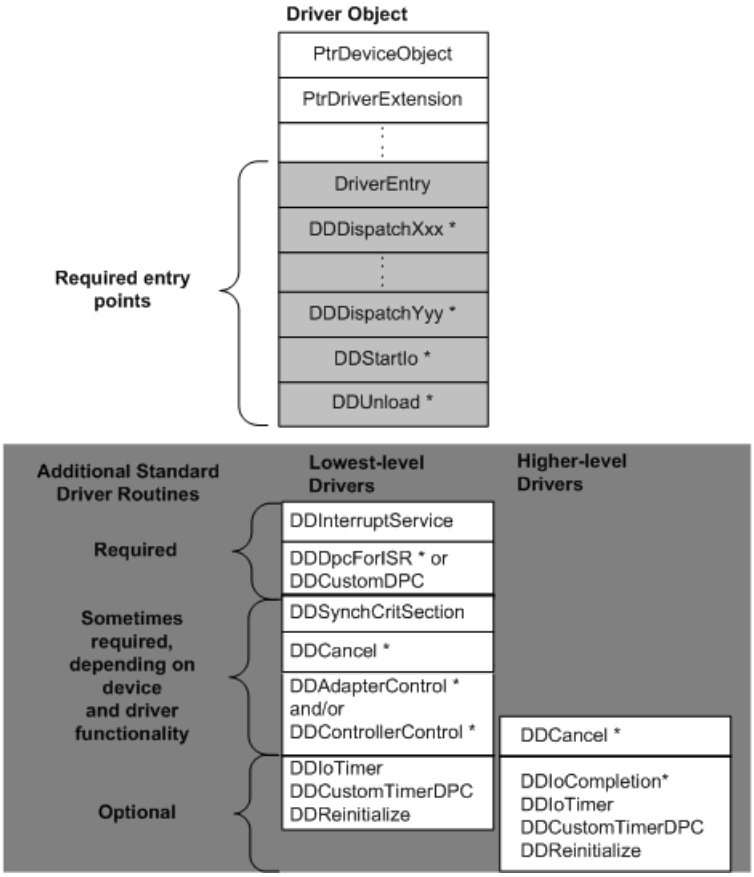
6/25/2019 • 2 minutes to read • Edit Online

The I/O manager creates a *driver object* for each driver that has been installed and loaded. Driver objects are defined using **DRIVER_OBJECT** structures.

When the I/O manager calls a driver's **DriverEntry** routine, it supplies the address of the driver's driver object. The driver object contains storage for entry points to many of a driver's standard routines. The driver is responsible for filling in these entry points.

The following figure illustrates a driver object, with the set of system-defined standard routines that lowest-level and higher-level drivers can or must have.

Each standard routine with an asterisk beside its name receives an I/O request packet (IRP) as input. Each of these standard routines also receives a pointer to the target device object for the I/O request.



The I/O manager defines the driver object type and uses driver objects to register and track information about the loaded images of drivers. Note that the dispatch entry points (**DDDispatch***Xxx* through **DDDispatch***Yyy*) in the driver object correspond to the major function codes (**IRP_MJ_*XXX***) that are passed in the I/O stack locations of IRPs.

The I/O manager routes each IRP first to a driver-supplied dispatch routine. A lowest-level driver's dispatch routine usually calls an I/O support routine (**IoStartPacket**) to queue (or pass on) each IRP that has valid arguments to the driver's *StartIo* routine. The *StartIo* routine starts the requested I/O operation on a particular device. Higher-level drivers usually do not have *StartIo* routines, but they can.

Driver Entry Points in Driver Objects

6/25/2019 • 2 minutes to read • Edit Online

A kernel-mode driver must specify the following entry points in its driver object:

- At least one dispatch routine's entry point, in order to get IRPs requesting PnP, power, and I/O operations.
- The entry point of its *AddDevice* routine, at **DriverObject -> DriverExtension -> AddDevice**.
- The entry point of its *StartIo* routine, if it manages its own queue of IRPs.
- If the driver can be loaded and/or replaced dynamically, an *Unload* entry point in order to free any system resources, such as system objects or memory, that the driver has allocated. (Drivers that cannot be replaced while the system is running, such as a keyboard driver, need not supply an *Unload* routine.)

These requirements do not apply to some miniport drivers, for which the corresponding class or port driver defines the entry points in the driver object. See the device-type-specific documentation for details.

The I/O manager maintains information about driver-created device objects in the corresponding driver object.

When a driver is loaded, its **DriverEntry** routine is called with a pointer to the driver object. When a driver's **DriverEntry** routine is called, it sets *Dispatch*, *StartIo* (if any), and *Unload* (if any) entry points directly in the driver object as follows:

```
DriverObject->MajorFunction[IRP_MJ_xxx] = DDDispatchXxx;
               :    :
DriverObject->MajorFunction[IRP_MJ_yyy] = DDDispatchYyy;
               :    :
DriverObject->DriverStartIo = DDStartIo;
DriverObject->DriverUnload = DDUnload;
               :    :
```

The **DriverEntry** routine also sets the entry point of its *AddDevice* routine, in the **DriverExtension** of its driver object, as follows:

```
DriverObject->DriverExtension->AddDevice = DDAddDevice;
```

A **DriverEntry** or optional *Reinitialize* routine also can use a field in the driver object (not shown in the driver object illustration) to get information from and/or set information in the configuration manager's registry database. For more information, see Registry Keys for Drivers.

The I/O manager exports no support routines to manipulate driver objects, which are **DRIVER_OBJECT** structures. Driver objects are used by the I/O manager to keep track of currently loaded drivers. Some members of a driver object are used only by the I/O manager. Others members are also used by driver writers; for example, you must know certain member names to define *AddDevice*, *Dispatch*, *StartIo*, and *Unload* entry points. You should neither attempt to use undocumented members within a **DRIVER_OBJECT** structure, nor make assumptions about the locations of any driver object members that are named in this documentation. Otherwise, you cannot maintain the portability of a driver from one Windows platform to another.

Other Standard Driver Routines

12/5/2018 • 2 minutes to read • Edit Online

As the driver object illustration shows, kernel-mode drivers have other standard routines along with those for which they set entry points in their respective driver objects. Most standard driver routines and some of the configuration-dependent objects they use are defined by the I/O manager. The ISR, *SynchCritSection* routine, and those shown in the Driver Object figure with names containing the word "custom" are defined by the NT kernel.

Most drivers use the device extension of each device object they create to maintain device-specific state about their I/O operations and to store pointers to any system resources that they must allocate in order to have other standard routines. For example, the **DDCustomTimerDpc** routine shown in the Driver Object figure requires the driver to supply storage for kernel-defined timer and DPC objects.

The set of standard driver routines for lowest-level drivers shown on the left in the driver object illustration is necessarily different from the set for higher-level drivers. Some of the routines shown in this figure are device-dependent or configuration-dependent requirements. Others are optional: you may choose to implement such a routine depending on the nature or configuration of the driver's devices, on the driver's design, and on the driver's position in a chain of layered drivers.

Writing a DriverEntry Routine

6/25/2019 • 2 minutes to read • Edit Online

Each driver must have a **DriverEntry** routine, which initializes driver-wide data structures and resources. The I/O manager calls the **DriverEntry** routine when it loads the driver.

In a driver that supports Plug and Play (PnP), as all drivers should, the **DriverEntry** routine is responsible for *driver* initialization, while the *AddDevice* routine (and, possibly, the dispatch routine that handles a PnP **IRP_MN_START_DEVICE** request) is responsible for *device* initialization. Driver initialization includes exporting the driver's other entry points, initializing certain objects the driver uses, and setting up various per-driver system resources. (Non-PnP drivers have significantly different requirements, as described in the Driver Development Kit [DDK] for Microsoft Windows NT 4.0 and earlier.)

DriverEntry routines are called in the context of a system thread at IRQL = PASSIVE_LEVEL.

A **DriverEntry** routine can be pageable and should be in an INIT segment so it will be discarded. Use an **alloc_text** pragma directive, as illustrated in the sample drivers that are provided with the Windows Driver Kit (WDK).

This section contains the following topics:

DriverEntry's Required Responsibilities

6/25/2019 • 2 minutes to read • Edit Online

The required, ordered responsibilities of a **DriverEntry** routine are as follows:

1. Supply entry points for the driver's standard routines.

   The driver stores entry points for many of its standard routines in the driver object or driver extension. Such entry points include those for the driver's *AddDevice* routine, dispatch routines, *StartIo* routine, and *Unload* routine. For example, a driver would set the entry points for its *AddDevice*, *DispatchPnP*, and *DispatchPower* routines with statements like the following (*Xxx* is a placeholder for a vendor-supplied prefix identifying the driver):

   ```
       :
   DriverObject->DriverExtension->AddDevice = XxxAddDevice;
   DriverObject->MajorFunction[IRP_MJ_PNP] = XxxDispatchPnp;
   DriverObject->MajorFunction[IRP_MJ_POWER] = XxxDispatchPower;
       :
   ```

   Additional standard routines, such as ISRs or *IoCompletion* routines, are specified by calling system support routines. For more information, see the descriptions of individual standard driver routines.

2. Create and/or initialize various driver-wide objects, types, or resources the driver uses. Note that most standard routines use objects on a per-device basis, so drivers should set up such objects in their *AddDevice* routines or after receiving an **IRP_MN_START_DEVICE** request.

   If the driver has a device-dedicated thread or waits on any kernel-defined dispatcher objects, the **DriverEntry** routine might initialize kernel dispatcher objects. (Depending on how the driver uses the object(s), it might instead perform this task in its *AddDevice* routine or after receiving an **IRP_MN_START_DEVICE** request.)

3. Free any memory that it allocated and is no longer required.

4. Return NTSTATUS indicating whether the driver successfully loaded and can accept and process requests from the PnP manager to configure, add, and start its devices. (See DriverEntry Return Values.)

DriverEntry's Optional Responsibilities

6/25/2019 • 2 minutes to read • Edit Online

Depending on the position of a particular driver in a chain of layered drivers, the nature of the underlying device, and the design of the driver, a **DriverEntry** routine also can be responsible for the following:

- Calling **IoAllocateDriverObjectExtension** to create and initialize a driver object extension, if the driver requires storage for data on a driver-wide basis. The driver object extension is a driver-specific data structure. For example, a driver might use its driver object extension to store a registry path or other global information.
- Calling **PsCreateSystemThread** to create executive worker threads, if the driver is a highest-level driver (such as a file system driver) that uses such threads. In this case, the driver must also have a callback routine of type WORKER_THREAD_ROUTINE, which takes a single input PVOID *Parameter*.
- Registering a *Reinitialize* routine. (See Writing a Reinitialize Routine.)
- Handling class-specific initialization requirements that differ from those discussed here, such as those that a device-specific miniport or miniclass driver working in tandem with a port or class driver might have. See the device-type specific documentation in the Windows Driver Kit (WDK) for details.

Providing Storage for System Resources

Per-device objects should be allocated in the *AddDevice* routine or in the Dispatch routine that handles the PnP **IRP_MN_START_DEVICE** request, not in **DriverEntry**.

However, a driver might need to allocate additional system-space memory for other driver-wide uses. If so, the **DriverEntry** routine can call one (or more) of the following routines:

- **IoAllocateDriverObjectExtension**, to create a context area associated with the driver object
- **ExAllocatePoolWithTag** for paged or nonpaged system-space memory
- **MmAllocateNonCachedMemory** or **MmAllocateContiguousMemory** for cache-aligned nonpaged system-space memory (used for I/O buffers)

Every **DriverEntry** routine is run in the context of a system thread at IRQL = PASSIVE_LEVEL. Therefore, any memory allocated with **ExAllocatePoolWithTag** for use exclusively during initialization can be from paged pool, as long as the driver does not control the device that holds the system page file. The allocated memory must be released with **ExFreePool** before **DriverEntry** returns control. However, a driver that sets a *Reinitialize* routine can pass a pointer to this memory when it calls **IoRegisterDriverReinitialization**, thus making the driver's *Reinitialize* routine responsible for freeing the memory allocation.

Claiming Hardware Resources

Older, non-PnP drivers claimed resources from the registry. PnP drivers, on the other hand, neither claim device resources from nor directly write resource requirements to the registry. Instead, these drivers report requirements in response to certain PnP IRPs, as part of the PnP manager's enumeration process. A PnP driver receives its allocated resources in a PnP **IRP_MN_START_DEVICE** request.

Drivers that do not interact directly with the PnP manager, such as certain miniport drivers, might have different reporting requirements imposed by a class or port driver that does interact with the PnP manager. Such requirements are specific to the device class. For device-specific and class-specific details, see the documentation for the relevant device class in the Windows Driver Kit (WDK).

Using the Registry

A **DriverEntry** routine might use the registry to get some of the information it needs to initialize the driver, or it might set information in the registry for other drivers or protected subsystems to use. The nature of the information depends on the type of device. Drivers can access the registry using **Zw*Xxx*** and **Rtl*Xxx*** routines. The **DriverEntry** routine's *RegistryPath* parameter points to a counted Unicode string that specifies a path to the driver's registry key, **\Registry\Machine\System\CurrentControlSet\Services*DriverName**. *The routine should save a copy of the string, not the pointer itself, since the pointer is no longer valid after *DriverEntry* returns.

DriverEntry Return Values

6/25/2019 • 2 minutes to read • Edit Online

A **DriverEntry** routine returns an NTSTATUS value, either STATUS_SUCCESS or an appropriate error status.

The **DriverEntry** routine should postpone any call to **IoRegisterDriverReinitialization** until just before it returns STATUS_SUCCESS. It must not make this call unless it will return STATUS_SUCCESS.

If a **DriverEntry** routine returns an NTSTATUS value that is not a success or informational value, such as STATUS_SUCCESS, the driver for that **DriverEntry** routine is not loaded.

A **DriverEntry** routine that will fail initialization must free any system objects, system resources, and registry resources it has already set up before it returns control. It should reset the driver's dispatch entry points in the driver object for **IRP_MJ_FLUSH_BUFFERS** and **IRP_MJ_SHUTDOWN** to **NULL** if the driver supports these requests.

If a driver will fail initialization, the **DriverEntry** routine also should log an error before returning control. See Logging Errors.

Note that a driver's *Unload* routine is not called if a driver's **DriverEntry** routine returns a failure status.

Writing a Reinitialize Routine

6/25/2019 • 2 minutes to read • Edit Online

Any driver that needs to initialize itself in stages can contain a *Reinitialize* routine. A *Reinitialize* routine is called after the **DriverEntry** routine has returned control and other drivers have initialized themselves. Typically, the *Reinitialize* routine performs tasks that must be done after another driver starts.

For example, the system's keyboard class driver,**kbdclass**, supports both PnP and legacy keyboard ports. If a system includes one or more legacy ports that the PnP manager cannot detect, the keyboard class driver must nevertheless create a device object for each port and layer itself over lower-level drivers for the port. Consequently, the class driver has a *Reinitialize* routine to be called after its **DriverEntry** and *AddDevice* routines have been called and other drivers have been loaded. The *Reinitialize* routine detects the port, creates a device object for it, and layers the driver over other lower-level drivers for the device.

A driver's **DriverEntry** routine calls **IoRegisterDriverReinitialization** to queue a *Reinitialize* routine for execution. The *Reinitialize* routine can also call **IoRegisterDriverReinitialization** itself, which causes the routine to be requeued. One of the parameters to *Reinitialize* indicates the number of times it has been called.

The call to **IoRegisterDriverReinitialization** can include a pointer to driver-defined context data, which the system supplies as input to *Reinitialize*. If the *Reinitialize* routine uses the registry, the context data should include the *RegistryPath* pointer that was passed to the **DriverEntry** routine because this pointer is not an input parameter to the *Reinitialize* routine.

The *Reinitialize* routine will not be called if **DriverEntry** does not return STATUS_SUCCESS.

Usually, a driver with a *Reinitialize* routine is a higher-level driver that controls both PnP and legacy devices. In addition to creating device objects for the devices that the PnP manager detects (and for which the PnP manager calls the driver's *AddDevice* routine), the driver must also create device objects for legacy devices that the PnP manager does not enumerate. The *Reinitialize* routine creates those device objects and layers the driver over the next-lower driver for the underlying device.

If a driver has a *Reinitialize* routine, it initializes in the same basic steps described in Writing a DriverEntry Routine, and it also has the same basic requirements as its **DriverEntry** routine.

Writing an AddDevice Routine

6/25/2019 • 2 minutes to read • Edit Online

Any driver that supports PnP must have an *AddDevice* routine. The *AddDevice* routine creates one or more device objects representing the physical, logical, or virtual devices for which the driver carries out I/O requests. It also attaches the device object to the device stack, so the device stack will contain a device object for each driver associated with the device.

The PnP manager calls a driver's *AddDevice* routine for each device controlled by the driver. *AddDevice* routines are called during system initialization (when devices are first enumerated), and any time a new device is enumerated while the system is running.

This section contains the following topics:

AddDevice Routines in Function or Filter Drivers

6/25/2019 • 3 minutes to read • Edit Online

An *AddDevice* routine in a function or filter driver should take the following steps:

1. Call **IoCreateDevice** to create a functional or filter device object (an FDO or filter DO) for the device being added.

   Do not specify a *DeviceName* for the device object, because doing so bypasses the PnP manager's security. If a user-mode component needs a symbolic link to the device, register a device interface (see the next step below). If a kernel-mode component needs a legacy device name, the driver must name the device object, but naming is not recommended.

   Include FILE_DEVICE_SECURE_OPEN in the *DeviceCharacteristics* parameter. This characteristic directs the I/O manager to perform security checks against the device object for all open requests, including relative opens and trailing file name opens.

2. [optional] Create one or more symbolic links to the device.

   Call **IoRegisterDeviceInterface** to register device functionality and create a symbolic link that applications or system components can use to open the device. The driver should enable the interface by calling **IoSetDeviceInterfaceState** when it handles the **IRP_MN_START_DEVICE** request. For more information, see Device Interface Classes.

3. Store the pointer to the device's PDO in the device extension.

   The PnP manager supplies a pointer to the PDO as the *PhysicalDeviceObject* parameter to *AddDevice*. Drivers use the PDO pointer in calls to routines such as **IoGetDeviceProperty**.

4. Define flags in the device extension to track certain PnP states of the device, such as device paused, removed, and surprise-removed.

   For example, define one flag to indicate that incoming IRPs should be held while the device is in a paused state. Create a queue for holding IRPs, if the driver does not already have a mechanism for queuing IRPs. See Queuing and Dequeuing IRPs for more information.

   Also allocate an **IO_REMOVE_LOCK** structure in the device extension and call **IoInitializeRemoveLock** to initialize this structure. For more information, see Using Remove Locks.

5. Set the DO_BUFFERED_IO or DO_DIRECT_IO flag bit in the device object to specify the type of buffering that the I/O manager is to use for I/O requests that are sent to the device stack. Higher-level drivers OR this member with the same value as the next-lower driver in the stack, except possibly for highest-level drivers. For more information, see Initializing a Device Object.

6. Set the DO_POWER_INRUSH or DO_POWER_PAGABLE flag for power management, if necessary. Drivers that are pageable must set the DO_POWER_PAGABLE flag. The device object flags are typically set by the bus driver when it creates the PDO for the device. However, higher-level drivers may occasionally need to alter the values of these flags in their *AddDevice* routines when they create the FDO or filter DO. See Setting Device Object Flags for Power Management for details.

7. Create and/or initialize any other software resources the driver uses to manage this device, such as events, spin locks, or other objects. (Hardware resources, such as I/O ports, are configured later, in response to an **IRP_MN_START_DEVICE** request.)

   Because an *AddDevice* routine runs in a system thread context at IRQL = PASSIVE_LEVEL, any memory

allocated with **ExAllocatePoolWithTag** for use exclusively during initialization can be from paged pool, as long as the driver does not control the device that holds the system page file. Such a memory allocation must be released with **ExFreePool** before *AddDevice* returns control.

8. Attach the device object to the device stack (**IoAttachDeviceToDeviceStack**).

   Specify a pointer to the device's PDO in the *TargetDevice* parameter.

   Store the pointer returned by **IoAttachDeviceToDeviceStack**. This pointer, which points to the device object of the next-lower driver for the device, is a required parameter to **IoCallDriver** and **PoCallDriver** when passing IRPs down the device stack.

9. Clear the DO_DEVICE_INITIALIZING flag in the FDO or filter DO with a statement like the following:

   ```
   FunctionalDeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
   ```

10. Be prepared to handle PnP IRPs for the device (such as **IRP_MN_QUERY_RESOURCE_REQUIREMENTS** and **IRP_MN_START_DEVICE**).

A driver must not start controlling the device until it receives an **IRP_MN_START_DEVICE** containing the list of hardware resources assigned to the device by the PnP manager.

AddDevice Routines in Bus Drivers

12/5/2018 • 2 minutes to read • Edit Online

A PnP bus driver has an *AddDevice* routine, but it is called when the bus driver is acting as the function driver for its controller or adapter. For example, the PnP manager calls the USB Hub bus driver's *AddDevice* routine to add the hub device. The hub driver's *AddDevice* routine is not called for a child of the hub (a device that plugs into the hub).

Guidelines for Writing AddDevice Routines

6/25/2019 • 2 minutes to read • Edit Online

Consider the following design guidelines when writing an *AddDevice* routine:

- If a filter driver determines its *AddDevice* routine was called for a device it does not need to service, the filter driver must return STATUS_SUCCESS to allow the rest of the device stack to be loaded for the device. The filter driver does not create a device object nor attach it to the device stack; the filter driver just returns success and allows the rest of the drivers to be added to the stack.

- A driver must provide storage, usually in the device extension of a device object, for any kernel-defined objects and executive spin locks it uses. A driver also must provide storage for pointers to certain objects obtained from the I/O manager or other system components.

  You might decide to allocate additional system-space memory for the driver's needs, such as for long-term I/O buffers or a lookaside list. If so, an *AddDevice* routine can call the following routines:

  ExAllocatePoolWithTag for paged or nonpaged system-space memory

  ExInitializePagedLookasideList or **ExInitializeNPagedLookasideList** to initialize a paged or nonpaged lookaside list

- If the driver has a device-dedicated thread or waits on any kernel-defined dispatcher objects, the *AddDevice* routine might initialize kernel dispatcher objects.

- If the driver uses any executive spin locks or provides the storage for an interrupt spin lock, the *AddDevice* routine might initialize these spin locks. See Spin Locks for more information.

- Tighten file-open security when calling **IoCreateDevice**.

  Specify the FILE_DEVICE_SECURE_OPEN characteristic on the call to **IoCreateDevice**. This characteristic is supported on Windows NT 4.0 SP5 and later. It directs the I/O manager to perform security checks against the device object for all open requests. Vendors should specify this characteristic on calls to **IoCreateDevice** if the FILE_DEVICE_SECURE_OPEN characteristic is not set in the device's class-installer INF or the device's INF and the drivers do not perform their own security checks on opens. (For more information, see Controlling Device Namespace Access.)

  If a driver sets the FILE_DEVICE_SECURE_OPEN characteristic when it calls **IoCreateDevice**, the I/O manager applies the security descriptor of the device object to any relative opens or trailing-filename opens. For example, if FILE_DEVICE_SECURE_OPEN is set for \Device\foo, and if \Device\foo can only be opened by the administrator, then \Device\foo\abc can also be opened by the administrator. The I/O manager, however, prevents a normal user from opening \Device\foo and \Device\foo\abc.

  If one driver for a device sets this characteristic, the PnP manager propagates it to all the device objects for the device.

Writing Dispatch Routines

6/25/2019 • 2 minutes to read • Edit Online

Processing any I/O request packet (IRP) begins in a dispatch routine that the driver registers to handle an IRP major function code (**IRP_MJ_*XXX**). *The driver's DriverEntry routine exports entry points for dispatch routines in a dispatch table within the driver's DRIVER_OBJECT** structure.

A driver can provide a separate dispatch routine for each major I/O function code that it handles. Alternatively, dispatch routines can be written to handle multiple I/O function codes.

This section contains the following topics:

Dispatch Routine Functionality

12/5/2018 • 2 minutes to read • Edit Online

The required functionality of a particular dispatch routine varies, depending on the I/O function code it handles, on the individual driver's position in a chain of drivers, and on the type of underlying physical device.

Most dispatch routines process incoming I/O request packets (IRPs) as follows:

1. Check the driver's I/O stack location in the IRP to determine what to do and check the parameters, if any, for validity.

   Whether a driver must check its I/O stack location to determine what to do and to check parameters depends on the given **IRP_MJ**_*XXX*, as well as on whether that driver set up a separate Dispatch routine for each **IRP_MJ**_*XXX* that the driver handles.

2. Satisfy the request and complete the IRP if possible; otherwise, pass it on for further processing by lower-level drivers or by other device driver routines.

   Whether a driver must pass on an IRP for further processing depends on the validity of the parameters, if any, as well as on the **IRP_MJ**_*XXX* and on the driver's level, if any, in a chain of layered drivers.

For more information about IRPs, see Handling IRPs.

Required Dispatch Routines

6/25/2019 • 2 minutes to read • Edit Online

Most drivers must handle the following *Dispatch* routines:

- *DispatchPnP*

  IRP_MJ_PNP indicates a request involving PnP device recognition, hardware configuration, or resource allocation. Such requests are typically sent to a device driver from the PnP manager or from a closely coupled higher-level driver.

- *DispatchPower*

  IRP_MJ_POWER indicates a request pertaining to the power state of either the device or the system. Such requests are sent to the device driver by either the power manager or a closely coupled higher-level driver.

- *DispatchCreate*

  IRP_MJ_CREATE indicates either that a user-mode protected subsystem, possibly on behalf of an application or subsystem-specific driver, has requested a handle for the file object associated with the target device object, or that a higher-level driver is connecting or attaching its device object to the target device object.

- *DispatchClose*

  IRP_MJ_CLOSE indicates that the last handle of the file object that was associated the target device object has been closed and released. All I/O requests have been completed or canceled, so there are no outstanding references to the file object pointer.

- *DispatchRead*

  IRP_MJ_READ indicates an I/O request to transfer data from the underlying physical device to the system.

- *DispatchWrite*

  IRP_MJ_WRITE indicates an I/O request to transfer data from the system to the underlying physical device.

- *DispatchDeviceControl*

  IRP_MJ_DEVICE_CONTROL indicates a request that contains a system-defined, device-type-specific I/O control code specifying a device type-specific operation. Higher-level drivers pass these IRPs on to their underlying device drivers, which typically process the request by accessing the device.

- *DispatchInternalDeviceControl*

  IRP_MJ_INTERNAL_DEVICE_CONTROL indicates a request sent to the device driver, in most cases from a closely coupled higher-level driver, usually with a privately defined, driver-specific and device-type-specific or device-specific I/O control code requesting a device-type-specific or device-specific operation.

  Only certain kinds of drivers are required to handle system-defined internal device I/O control requests, including certain SCSI drivers, keyboard or mouse device drivers, and parallel drivers that interoperate with system-supplied drivers.

- *DispatchSystemControl*

  IRP_MJ_SYSTEM_CONTROL is used to specify WMI requests to drivers. For more information about

WMI, see Windows Management Instrumentation.

The dispatch routines that a driver must provide vary according to the type and functionality of the underlying physical device. For device-type-specific information about IRP major function codes that drivers must handle, see the device-type specific documentation in the Windows Driver Kit (WDK).

Optional Dispatch Routines

6/25/2019 • 2 minutes to read • Edit Online

Drivers might include the following dispatch routines:

- *DispatchCleanup*

  IRP_MJ_CLEANUP indicates that the last handle for a file object that is associated with the target device object is being closed. Outstanding I/O requests for the file object might still exist. Drivers can implement a *DispatchCleanup* routine to perform cleanup that is not specific to any particular file handle. Drivers can also use their *DispatchClose* routine for the same purpose.

- *DispatchQueryInformation*, *DispatchSetInformation*

  Some highest-level drivers might have to process **IRP_MJ_QUERY_INFORMATION** and **IRP_MJ_SET_INFORMATION** IRPs. Such requests indicate that a user-mode application, kernel-mode component, or driver has requested information about the length of the file object (representing the driver's device object) for which the user-mode requester has a handle, or that the user-mode requester is attempting to set an end-of-file on that file object.

  Parallel class and serial device drivers handle these requests by setting the **FILE_STANDARD_INFORMATION** or **FILE_POSITION_INFORMATION** length or position to zero. Other highest-level device drivers should support these requests, particularly if a user-mode application or kernel-mode driver might call C runtime functions to manipulate the file object. File system drivers must support these requests more fully than these highest-level device drivers.

- *DispatchFlushBuffers*

  A driver that caches data in a device or buffers data internally in driver-allocated memory might receive **IRP_MJ_FLUSH_BUFFERS**. Receipt of this request indicates that the driver should write its buffered data or flush the cached data out to the device, or should discard buffered or cached data that was read from the device.

  For example, the system keyboard and mouse class drivers, which have internal ring buffers for input data from their devices, support the flush request. Drivers of mass-storage devices and drivers layered above them also support this request.

- *DispatchShutdown*

  Any driver that is likely to be called before the system shuts down must handle **IRP_MJ_SHUTDOWN**. The *DispatchShutdown* routine should do whatever driver-determined cleanup is necessary before the power manager sends a system set-power IRP to shut down the system. A driver can call **IoRegisterShutdownNotification** or **IoRegisterLastChanceShutdownNotification** to register for shutdown notification.

Drivers for mass-storage devices and intermediate drivers layered over them can rely on a highest-level file system driver to send them shutdown IRPs when the system is about to shut down. That is, the FSD is responsible for making sure that any cached file data is written out to peripheral devices, calling underlying drivers to flush data from their device caches or buffers (if any), and so forth before the system is shut down.

The driver of a mass-storage device that caches data internally must provide *DispatchShutdown* and *DispatchFlushBuffers* routines. If a mass-storage driver buffers data in memory but its device has no internal cache, it also must provide *DispatchShutdown* and *DispatchFlushBuffers* routines.

Any intermediate driver layered above a driver that handles **IRP_MJ_FLUSH_BUFFERS** and **IRP_MJ_SHUTDOWN** requests also provide *DispatchShutdown* and *DispatchFlushBuffers* routines.

Dispatch Routines and IRQLs

6/25/2019 • 2 minutes to read • Edit Online

Most drivers' dispatch routines are called in an arbitrary thread context at IRQL = PASSIVE_LEVEL, with the following exceptions:

- Any highest-level driver's dispatch routines are called in the context of the thread that originated the I/O request, which is commonly a user-mode application thread.

  In other words, the dispatch routines of file system drivers and other highest-level drivers are called in a nonarbitrary thread context at IRQL = PASSIVE_LEVEL.

- The *DispatchRead*, *DispatchWrite*, and *DispatchDeviceControl* routines of lowest-level device drivers, and of intermediate drivers layered above them in the system paging path, can be called at IRQL = APC_LEVEL and in an arbitrary thread context.

  The *DispatchRead* and/or *DispatchWrite* routines, and any other routine that also processes read and/or write requests in such a lowest-level device or intermediate driver, must be resident at all times. These driver routines can neither be pageable nor be part of a driver's pageable-image section; they must not access any pageable memory. Furthermore, they should not be dependent on any blocking calls (such as **KeWaitForSingleObject** with a nonzero time-out).

- The *DispatchPower* routine of drivers in the hibernation and/or paging paths can be called at IRQL = DISPATCH_LEVEL. The *DispatchPnP* routines of such drivers must be prepared to handle PnP **IRP_MN_DEVICE_USAGE_NOTIFICATION** requests.

- The *DispatchPower* routine of drivers that require inrush power at start-up can be called at IRQL = DISPATCH_LEVEL.

For additional information, see Managing Hardware Priorities.

When to Check the Driver's I/O Stack Location

6/25/2019 • 2 minutes to read • Edit Online

A major I/O function code is set in the driver's I/O stack location for each incoming IRP.

A driver's dispatch routine must check the driver's I/O stack location for the IRP to determine what to do if any of the following conditions hold:

- The dispatch routine handles more than one major I/O function code.
- The dispatch routine must handle a set of minor function codes for certain major function codes. IRPs with minor function codes include **IRP_MJ_PNP** and **IRP_MJ_POWER**, as well as certain IRPs that the SCSI port driver and file system drivers must handle.
- The dispatch routine of a device driver or of a closely coupled higher-level driver handles **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests, which have an associated set of I/O control codes.

To get a pointer to a driver's I/O stack location, its dispatch routine calls **IoGetCurrentIrpStackLocation**.

Higher-level drivers' dispatch routines always call **IoGetCurrentIrpStackLocation** and also call **IoGetNextIrpStackLocation** to get a pointer to the next-lower driver's I/O stack location for IRPs that they set up for the next-lower driver, when passing IRPs down the driver stack.

The *DispatchDeviceControl* routine or *DispatchInternalDeviceControl* routine of a device driver, or possibly of its closely coupled class driver(s), must determine which I/O control code is set in the driver's I/O stack location at **Parameters.DeviceIoControl.IoControlCode** for each request. The I/O control code is contained in the driver's I/O stack location.

In most cases, the *DispatchDeviceControl* or *DispatchInternalDeviceControl* routine of a higher-level driver simply passes an **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL** request on to the next-lower driver, after setting up its stack location in the IRP. However, SCSI class drivers must check for certain SCSI Port I/O control codes so that they can set up the SCSI port driver's I/O stack location correctly before passing on these requests.

DispatchCreate, DispatchClose, and DispatchCreateClose Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *DRIVER_DISPATCH* IRPs with I/O function codes of **IRP_MJ_CREATE** and **IRP_MJ_CLOSE**, respectively. Alternatively, a combined *DispatchCreateClose* routine can handle IRPs for both of these I/O function codes.

A create request can originate either from a user-mode subsystem's attempt to get a handle to a file object representing a device (possibly on behalf of an application or subsystem-level driver) or in a higher-level driver's call to **IoGetDeviceObjectPointer** or **IoAttachDevice**.

A reciprocal close request originates from a user-mode subsystem's close of the file object handle associated with the driver's device object.

Each of these requests is inherently synchronous.

Separate DispatchCreate and DispatchClose Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *Dispatch* routines for **IRP_MJ_CREATE** and **IRP_MJ_CLOSE** requests might do nothing more than complete the input IRP with STATUS_SUCCESS. For more information, see Completing IRPs.

Another driver's *Dispatch* routines for **IRP_MJ_CREATE** and **IRP_MJ_CLOSE** requests might do more work, depending on the underlying device driver or on the underlying device. Consider the following scenarios:

- On receipt of a create request, a class driver might initialize an internal queue and send an **IRP_MJ_INTERNAL_DEVICE_CONTROL** request down to the corresponding port driver requesting device configuration information or exclusive access to a controller port.
- Receipt of **IRP_MJ_CLOSE** indicates that the last reference to the file object that is associated with the target device object has been removed. This implies that all handles to the file object has been closed and all outstanding I/O requests have been completed or canceled.
- On receipt of a create request, a driver of an infrequently used device might call **MmLockPagableCodeSection** to make resident some of the driver routines that process other **IRP_MJ_*XXX*** requests. On receipt of a reciprocal close request, the driver might call **MmUnlockPagableImageSection** to conserve system memory by having its pageable-image section paged out when all file object handles for such a driver's device object(s) are closed.

Some drivers handle **IRP_MJ_CLOSE** requests only for symmetry because, after their device objects have been opened by a protected subsystem or higher-level driver, the lower-level drivers' device objects are not closed until the system itself is shut down. For example, keyboard and mouse drivers set up device objects representing physical devices that must be functional while the system is running, so these drivers might have minimal *DispatchClose* routines for symmetry, or they might have combined *DispatchCreateClose* routines.

If the device controlled by a lower-level driver must be available for the system to continue running, the driver's *DispatchClose* routine generally will not be called. For example, some of the system disk drivers have no *DispatchClose* routine, but these drivers usually have *DispatchFlushBuffers* and *DispatchShutdown* routines to complete any outstanding file I/O operations before the system is shut down.

While you can implement separate *DRIVER_DISPATCH* and *DispatchClose* routines, drivers sometimes have a single DispatchCreateClose routine for handling both create and close requests.

A Single DispatchCreateClose Routine

6/25/2019 • 2 minutes to read • Edit Online

Many drivers, particularly lower-level drivers in a chain of layered drivers, merely need to establish their existence on receipt of a *create* request and merely need to acknowledge the receipt of a *close* request.

For example, a port driver for a device controller with one or more closely coupled class drivers that call **IoGetDeviceObjectPointer** might have a minimal *DispatchCreateClose* routine. The routine might do nothing more than complete the IRP as follows:

```
    :    :
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
 Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
 return STATUS_SUCCESS;
}
```

This minimal *DispatchCreateClose* routine sets the **Information** member of the I/O status block to zero, indicating the file object is opened for a create request; **Information** has no meaning for a close request. The routine sets the **Status** member to STATUS_SUCCESS and also returns this status value, indicating that the driver is ready to accept I/O requests.

This minimal *DispatchCreateClose* routine completes the create IRP without boosting the priority of the originator of the IRP (IO_NO_INCREMENT), because the originator is assumed to wait for an indeterminate but very small interval for the request to complete.

How much work a *DispatchCreateClose* routine does depends partly on the nature of the driver's device or the underlying device and partly on the design of the driver. If a driver performs very different operations for create and close requests, it should handle these requests in separate DispatchCreate and DispatchClose routines.

To handle a create request to open a file object representing a logical or physical device, a highest-level driver should do the following:

1. Call **IoGetCurrentIrpStackLocation** to get a pointer to its I/O stack location in the IRP.
2. Check **FileObject**.**FileName** in the I/O stack location and complete the IRP with STATUS_SUCCESS if the Unicode string at **FileName** has a zero length; otherwise, complete the IRP with STATUS_INVALID_PARAMETER.

Following the preceding steps ensures that no attempt to open a pseudofile on a device can cause problems later. For example, this prevents attempts to open a nonexistent \\device\parallel0\temp.dat.

Rules for Implementing DispatchCreate, DispatchClose, and DispatchCreateClose Routines

12/5/2018 • 2 minutes to read • Edit Online

Keep the following points in mind when implementing *DispatchCreate*, *DispatchClose*, and *DispatchCreateClose* routines:

- At a minimum, the routine must do the following:

  1. Set the **Status** field of the input IRP's I/O status block with an appropriate NTSTATUS, usually STATUS_SUCCESS.
  2. Set the **Information** field of the input IRP's I/O status block to zero.
  3. Call **IoCompleteRequest** with the IRP and a *PriorityBoost* of IO_NO_INCREMENT.
  4. Return the NTSTATUS that it set in the **Status** field of the IRP's I/O status block.

- In a highest-level or intermediate driver, the routine might have to do additional work to process a create or close request, depending on the nature of its device or of the underlying device, and on the design of the driver.

- For a create request to open a file object that represents a logical or physical device, a highest-level driver should check the **FileObject.FileName** in the I/O stack location and complete the IRP with STATUS_SUCCESS if the Unicode string at **FileName** has a zero length. Otherwise, it should complete the IRP with STATUS_INVALID_PARAMETER.

- The routines of lowest-level drivers are called only when the next-higher-level driver calls **IoAttachDeviceToDeviceStack**, **IoGetDeviceObjectPointer**, or **IoAttachDevice**. The lowest-level driver in a chain of layered drivers frequently does only the minimum required processing of a create or close request.

DispatchCleanup Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *DispatchCleanup* routine handles IRPs for the **IRP_MJ_CLEANUP** I/O function code.

Drivers can use a *DispatchCleanup* routine to perform any cleanup operations that are needed after all of the handles to a file object have been closed. Note that *DispatchCleanup* is called in the process context of the process that closed the final handle; this process might be different from the process that initially opened the handle. (Typically this difference happens because another process uses the **DuplicateHandle** user-mode routine to duplicate the processes handles.) Drivers that must perform cleanup in the original process context can use the **PsSetCreateProcessNotifyRoutine** routine to register a callback routine for that purpose, but keep in mind that such callbacks are a limited system resource.

In general, a *DispatchCleanup* routine must process an **IRP_MJ_CLEANUP** request by doing the following for every IRP that is currently in the device queue (or in the driver's internal queue of IRPs), for the target device object, and is associated with the file object:

- Call **IoSetCancelRoutine** to set the *Cancel* routine pointer to **NULL**.
- Cancel every IRP that is currently in the queue for the target device object, if the file object that is specified in the driver's I/O stack location of the queued IRP matches the file object that was received in the I/O stack location of the **IRP_MJ_CLEANUP** request.
- Call **IoCompleteRequest** to complete the IRP, and return STATUS_SUCCESS.

While processing an **IRP_MJ_CLEANUP** request, a driver can receive additional requests, such as **IRP_MJ_READ** or **IRP_MJ_WRITE**. Therefore, a driver that must deallocate resources must also synchronize execution of its *DispatchCleanup* routine with other dispatch routines, such as *DispatchRead* and *DispatchWrite*.

DispatchRead, DispatchWrite, and DispatchReadWrite Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *DispatchRead* and *DispatchWrite* routines handle IRPs with I/O function codes of **IRP_MJ_READ** and **IRP_MJ_WRITE**, respectively. Alternatively, a combined *DispatchReadWrite* routine can handle IRPs for both of these I/O function codes.

Every driver of a device from which data can be transferred to the system must have a *DispatchRead* routine. Every driver of a device to which data can be transferred from the system must have a *DispatchWrite* routine. Any driver that transfers data in both directions can have a combined *DispatchReadWrite* routine.

Lower-level drivers handle **IRP_MJ_READ** and **IRP_MJ_WRITE** requests asynchronously. Therefore, *DispatchRead* and/or *DispatchWrite* routines in highest-level drivers must pass these requests on for further processing, provided that the request has valid parameters in that driver's I/O stack location of the IRP.

Whether a driver sets up its device objects for buffered or direct I/O affects how it handles transfer requests. In particular, a driver that uses direct I/O to do DMA operations might need to split up large transfer requests into a sequence of smaller transfer operations in order to satisfy an **IRP_MJ_READ** or **IRP_MJ_WRITE** request. For more information, see Input/Output Techniques.

The following subsections discuss some of the design and implementation considerations for *DispatchReadWrite* routines in lowest-level device drivers that use buffered I/O and direct I/O, as well as in higher-level drivers layered above them:

Handling Transfers Asynchronously

6/25/2019 • 2 minutes to read • Edit Online

Except for highest-level drivers, all drivers handle **IRP_MJ_READ** and **IRP_MJ_WRITE** requests asynchronously. The *DispatchRead* and *DispatchWrite* routines in even a highest-level driver cannot wait for lower-level drivers to finish processing an asynchronous read or write request; they must pass such a request on to lower drivers and return STATUS_PENDING.

Similarly, a lowest-level device driver's *DispatchReadWrite* routine must pass the transfer request on to other driver routines that process device I/O requests and then return STATUS_PENDING.

A higher-level driver sometimes must set up partial-transfer IRPs and pass them on to lower drivers. The higher-level driver can complete the original read/write IRP only when its partial-transfer requests have been completed by the lower drivers.

For example, a SCSI class driver's *DispatchReadWrite* routine is required to split large transfer requests that exceed the underlying HBA's transfer capabilities into a set of partial-transfer requests. The class driver must set up the parameters in its partial-transfer IRPs so that the SCSI port/miniport drivers can satisfy each partial-transfer request in a single DMA operation.

Other device drivers that use DMA or PIO also might need to split up large transfer requests for themselves.

For more information about using DMA and PIO, see Input/Output Techniques.

DispatchReadWrite Using Buffered I/O

6/25/2019 • 2 minutes to read • Edit Online

Any lowest-level device driver that sets up its device objects for buffered I/O satisfies a read request by returning data transferred from its device into a locked down system-space buffer at **Irp->AssociatedIrp.SystemBuffer**. It satisfies a write request by transferring data from the same buffer out to its device.

Consequently, the *DispatchReadWrite* routine of such a device driver usually does the following on receipt of a transfer request:

1. Calls **IoGetCurrentIrpStackLocation** and determines the direction of the transfer request.
2. Checks the validity of the parameters for the request.
   - For a read request, the routine usually checks the driver's *IoStackLocation*->**Parameters.Read.Length** value to determine whether the buffer is large enough to receive data transferred from the device.

     For example, the system keyboard class driver processes read requests that come only from the Win32 user input thread. This driver defines a structure, KEYBOARD_INPUT_DATA, in which to store keystrokes from the device and, at any given moment, holds some number of these structures in an internal ring buffer in order to satisfy read requests as they come in.
   - For a write request, the routine usually checks the value at **Parameters.Write.Length**, and checks the data at **Irp->AssociatedIrp.SystemBuffer** for validity if necessary: that is, if its device accepts only structured data packets containing members with defined value ranges.
3. If any parameters are invalid, the *DispatchReadWrite* routine completes the IRP immediately, as already described in Completing IRPs. Otherwise, the routine passes the IRP on for further processing by other driver routines, as described in Passing IRPs down the Driver Stack.

Lowest-level device drivers that use buffered I/O usually must satisfy a transfer request by reading or writing data of a size specified by that the originator of the request. Such a driver is likely to define a structure for data coming in from or being sent to its device and is likely to buffer structured data internally, as the system keyboard class driver does.

Drivers that buffer data internally should support **IRP_MJ_FLUSH_BUFFERS** requests, and can also support **IRP_MJ_SHUTDOWN** requests.

The highest-level driver in a chain is usually responsible for checking the input IRP's parameters before passing a read/write request on to lower drivers. Consequently, many lower-level drivers can assume that their I/O stack locations in a read/write IRP have valid parameters. If a lowest-level driver in a chain is aware of device-specific constraints on data transfers, that driver is required to check the validity of the parameters in its I/O stack location.

DispatchReadWrite Using Direct I/O

6/25/2019 • 2 minutes to read • Edit Online

Any lower-level device driver that sets up its device objects for direct I/O satisfies a read request by returning data transferred from its device to system physical memory, which is described by the MDL at **Irp->MdlAddress**. It satisfies a write request by transferring data from system physical memory out to its device.

Lower-level drivers must handle read/write requests asynchronously. Therefore, every lower-level driver's *DispatchReadWrite* routine must pass **IRP_MJ_READ** and **IRP_MJ_WRITE** IRPs with valid parameters on to other driver routines, as described in Passing IRPs down the Driver Stack.

For read/write IRPs sent to lower-level drivers, the paged physical memory described by the MDL at **Irp->MdlAddress** has already been probed for the correct access rights to carry out the requested transfer and has already been locked down by the highest-level driver in the chain or by the I/O manager. Any intermediate or lowest-level driver that sets up its device objects for direct I/O should not call **MmProbeAndLockPages** because this has already been done. A lowest-level driver calls **MmGetSystemAddressForMdlSafe**. (Drivers for Windows 98 call **MmGetSystemAddressForMdl** instead. Drivers for Windows Me, Windows 2000 and later versions of Windows should use **MmGetSystemAddressForMdlSafe**.)

Any intermediate or lowest-level device driver's *DispatchReadWrite* routine should validate the parameters in its I/O stack location of read/write IRPs if it cannot trust a higher-level driver to pass down only IRPs with valid parameters. If the *DispatchReadWrite* routine finds a parameter error, it should complete the IRP with an appropriate error STATUS_*XXX* value as already described in Completing IRPs. If parameters are valid, an intermediate driver's *DispatchReadWrite* routine must pass the request on for further processing, according to the guidelines in DispatchReadWrite in Higher-Level Drivers.

A lowest-level device driver's *DispatchReadWrite* routine must call **IoMarkIrpPending** with the transfer request, pass the IRP on for further processing by other driver routines, and return STATUS_PENDING, as described in Passing IRPs down the Driver Stack.

Note that a device driver's *DispatchReadWrite* routine can control the order in which IRPs are queued to its device for faster I/O throughput by calling **IoStartPacket** with a driver-determined *Key* value. Another routine in the driver dequeues the IRP later, determines whether the requested length must be split into partial-transfer operations, and programs the device to transfer data.

In general, a device driver that must split up large transfer requests to suit the limitations of its device should postpone these operations until just before setting up the device for a given transfer request. Such a device driver's *DispatchReadWrite* routine should not check the I/O stack location of incoming IRPs for any device-specific transfer constraints, nor attempt to calculate partial-transfer ranges, when the driver can postpone these checks until just before its *StartIo* (or other driver routine) programs the device for a transfer operation.

DispatchReadWrite in Higher-Level Drivers

6/25/2019 • 2 minutes to read • Edit Online

Except for file system drivers, a higher-level driver usually does not have any internal driver queues for IRPs. Such a driver's *DispatchReadWrite* routine can pass IRPs with valid parameters on to lower drivers, possibly after setting up its *IoCompletion* routine, as described in Passing IRPs down the Driver Stack.

However, a SCSI class driver's *DispatchReadWrite* routine is responsible for splitting up large transfer requests, if necessary, before it sends an IRP with the major function code **IRP_MJ_READ** or **IRP_MJ_WRITE** to the SCSI port/miniport driver pair. For more information, see Storage Class Driver's SplitTransferRequest Routine.

If a higher-level driver allocates one or more IRPs, which it sets up for the next-lower driver in its *DispatchReadWrite* routine, to request some number of partial transfers, the *DispatchReadWrite* routine must call **IoSetCompletionRoutine** with each driver-allocated IRP. The driver must register its *IoCompletion* routine to track how much data is transferred in each partial-transfer operation so that the *IoCompletion* routine can release all driver-allocated IRPs and, eventually, complete the original request.

If the underlying driver controls a removable-media device, any IRPs allocated by the higher-level driver must have a thread context. To set up a thread context, the allocating driver must set the **Irp->Tail.Overlay**.Thread in each newly allocated IRP from the same value in the incoming transfer IRP. For more information, see Supporting Removable Media.

If the underlying device driver returns an IRP for a partial transfer with an error, the *IoCompletion* routine can either retry the partial-transfer request or complete the original IRP with its I/O status block set with the returned error, after freeing any IRPs and memory the higher-level driver has allocated.

If a higher-level driver's *DispatchReadWrite* routine allocates memory for partial-transfer operations and its allocation will be accessed by the driver's *IoCompletion* routine (or by the underlying device driver), the *DispatchReadWrite* routine must allocate that memory from nonpaged pool.

Summary of Read/Write Dispatch Routines

6/25/2019 • 2 minutes to read • Edit Online

Keep the following points in mind when implementing a *DispatchRead*, *DispatchWrite*, or *DispatchReadWrite* routine:

- It is the responsibility of the highest-level driver in a chain of layered drivers to check the parameters of incoming read/write IRPs for validity before setting up the next-lower-level driver's I/O stack location in an IRP.
- Intermediate and lowest-level drivers generally can rely on the highest-level driver in their chain to pass down transfer requests with valid parameters. However, any driver can perform sanity checks on the parameters in its I/O stack location of an IRP, and each device driver should check the parameters for conditions that might violate any restrictions imposed by its device.
- If a *DispatchReadWrite* routine completes an IRP with an error, it should set the I/O stack location **Status** member with an appropriate NTSTATUS-type value, set the **Information** member to zero, and call **IoCompleteRequest** with the IRP and a *PriorityBoost* of IO_NO_INCREMENT.
- If a driver uses buffered I/O, it might need to define a structure to contain data to be transferred and might need to buffer some number of these structures internally.
- If a driver uses direct I/O, it might need to check whether the MDL at **Irp->MdlAddress** describes a buffer containing too much data (or too many page breaks) for the underlying device to handle in a single transfer operation. If so, the driver must split up the original transfer request into a sequence of smaller transfer operations.

  A closely coupled class driver might split up such a request in its *DispatchReadWrite* routine for its underlying port driver. SCSI class drivers, particularly for mass-storage devices, are required to do this. For more information about requirements for SCSI drivers, see Storage Drivers.
- A lower-level device driver's *DispatchReadWrite* routine should postpone splitting a large transfer request into partial transfers until another driver routine dequeues the IRP to set up the device for the transfer.
- If a lower-level device driver queues a read/write IRP for further processing by its own routines, it must call **IoMarkIrpPending** before it queues the IRP. The *DispatchReadWrite* routine also must return control with STATUS_PENDING in these circumstances.
- If the *DispatchReadWrite* routine passes an IRP on to lower drivers, it must set up the I/O stack location for the next-lower driver in the IRP. Whether the higher-level driver also sets an *IoCompletion* routine in the IRP before passing it on with **IoCallDriver** depends on the design of the driver and of those layered under it.

  However, a higher-level driver must call **IoSetCompletionRoutine** before it calls **IoCallDriver** if it allocates any resources, such as IRPs or memory. Its *IoCompletion* routine must free any driver-allocated resources when lower drivers have completed the request but before the *IoCompletion* routine calls **IoCompleteRequest** with the original IRP.
- If a higher-level driver allocates IRPs for lower drivers that might include an underlying removable-media device driver, the allocating driver must establish the thread context in each IRP it allocates.

DispatchDeviceControl and DispatchInternalDeviceControl Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's dispatch routines (see **DRIVER_DISPATCH**) handle IRPs with I/O function codes of **IRP_MJ_DEVICE_CONTROL** and **IRP_MJ_INTERNAL_DEVICE_CONTROL**, respectively.

For every common type of peripheral device, the system defines a set of I/O control codes for **IRP_MJ_DEVICE_CONTROL** requests. New drivers for each type of device must support these requests. In most cases, these public I/O control codes for each type of device are not exported to user-mode applications.

Some of these system-defined I/O control codes are used by higher-level drivers that create IRPs for the underlying device driver by calling **IoBuildDeviceIoControlRequest**. Others are used by Win32 components to communicate with an underlying device driver by calling the Win32 function **DeviceIoControl** (described in Microsoft Windows SDK documentation) which, in turn, calls a system service. The I/O manager sets up an IRP, and stores the major function code **IRP_MJ_DEVICE_CONTROL** and the given I/O control code in the **IO_STACK_LOCATION** structure at **Parameters.DeviceIoControl.IoControlCode**. Then, the I/O manager calls the *DispatchDeviceControl* routine of the highest-level driver in the chain.

For certain system-supplied drivers designed to interoperate with and support new drivers, the operating system also defines a set of I/O control codes for **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests. In most cases, these public I/O control codes allow add-on higher-level drivers to interoperate with an underlying device driver.

As an example, the system-supplied parallel drivers support a set of internal I/O control codes that vendor-supplied drivers send in **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests. For more information, see Internal Device Control Requests for Parallel Ports and Internal Device Control Requests for Parallel Devices.

Almost all operations requested through system-defined I/O control codes use buffered I/O, because this type of request seldom requires the transfer of large amounts of data. That is, even drivers that set up their device objects for direct I/O are sent IRPs for device control requests with data to be transferred into or out of the buffer at **Irp->AssociatedIrp.SystemBuffer** (except for certain types of highest-level device drivers with closely coupled Win32 multimedia drivers).

In addition, a driver can define a set of private I/O control codes that other drivers can use to communicate with it. New public I/O control codes can be added to the system only with the cooperation of Microsoft Corporation, because public I/O control codes are built into the operating system itself.

For specific information about the set of public I/O control codes that different kinds of drivers must support and about defining private I/O control codes, see the device-specific reference sections of the Windows Driver Kit (WDK).

DispatchDeviceControl in Lowest-Level Drivers

6/25/2019 • 2 minutes to read • Edit Online

An **IRP_MJ_DEVICE_CONTROL** request for a lowest-level driver requires that the driver either change the state of its device or provide information about the state of its device. Because most kinds of drivers are required to handle a number of I/O control codes, their *DispatchDeviceControl* routines usually contain a **switch** statement somewhat like the following:

```
    :    :
switch (irpSp->Parameters.DeviceIoControl.IoControlCode)
{
    case IOCTL_DeviceType_XXX:
    case IOCTL_DeviceType_YYY:
        if (irpSp->Parameters.DeviceIoControl.InputBufferLength <
                (sizeof(IOCTL_XXXYYY_STRUCTURE)))
        {
            status = STATUS_BUFFER_TOO_SMALL;
            break;
        } else {
            IoMarkIrpPending(Irp);
     :      : // pass IRP on for further processing
    case ...
     :      :
```

As this code fragment shows, a *DispatchDeviceControl* routine also checks parameters, sometimes on each I/O control code that the driver must support, sometimes on groups of these I/O control codes.

Consider the following implementation guidelines for device drivers' *DispatchDeviceControl* routines:

- *DispatchDeviceControl* must check the parameters for validity, and immediately complete IRPs with parameter errors, as described in Completing IRPs.
- Grouping I/O control codes in a **case** statement (where practical) when testing for valid parameters is economical in terms of driver performance and size and in code maintenance. As the preceding code fragment suggests, I/O control codes that use a common structure are natural candidates for such a **case** group.
- Switching first on any I/O control codes for which the *DispatchDeviceControl* routine can satisfy and complete the IRP improves performance because the driver can return control faster.
- Switching later on I/O control codes that specify infrequently requested operations also can improve the driver's performance in processing **IRP_MJ_DEVICE_CONTROL** requests.
- For better performance, every lowest-level device driver's *DispatchDeviceControl* routine should satisfy any device control request that it can, without queuing the IRP to other driver routines.

If the *DispatchDeviceControl* routine can complete the IRP, it should call **IoCompleteRequest** with a *PriorityBoost* of IO_NO_INCREMENT. If the *DispatchDeviceControl* routine must queue the IRP for further processing, it must call **IoMarkIrpPending** and return STATUS_PENDING.

DispatchDeviceControl in Higher-Level Drivers

6/25/2019 • 2 minutes to read • Edit Online

Usually, the *DispatchDeviceControl* routine of a higher-level driver simply sets up the I/O stack location for the next-lower-level driver and passes the IRP on with **IoCallDriver**. The *DispatchDeviceControl* routine seldom checks the validity of parameters in the input IRP because the underlying device driver is assumed to have better information about how to handle each device-type-specific I/O control request.

A possible exception to this general rule is the *DispatchDeviceControl* routine in the class driver of a class/port driver pair. For more information about handling device control requests in paired class/port drivers, see Dispatch(Internal)DeviceControl in Class/Port Drivers.

Any new higher-level driver that is not closely associated with a particular device driver should simply set up the I/O stack location for the next-lower-level driver and pass the **IRP_MJ_DEVICE_CONTROL** request on for further processing.

A device control request is usually handled synchronously. That is, a higher-level driver's *DispatchDeviceControl* routine can frequently return control to the system as follows:

```
    :    :
return IoCallDriver(DeviceObject->NextDeviceObject, Irp);
```

However, a higher-level driver cannot use the preceding technique if a lower driver might return STATUS_PENDING for such a request. In that case, the higher-level driver should call **IoSetCompletionRoutine** to register an *IoCompletion* routine. When the *IoCompletion* routine is called, it can check the I/O status block to determine whether the IRP is still pending. If it is, the *IoCompletion* routine might retry the request or, possibly, call **IoMarkIrpPending** before it calls **IoCompleteRequest** and returns STATUS_PENDING. A higher-level driver must not complete an IRP with STATUS_PENDING unless it has called **IoMarkIrpPending** for that IRP first.

If the underlying device driver must process much data transferred from the device before it completes the request, then a higher-level driver might handle such a device control request asynchronously. That is, the higher-level driver might call **IoSetCompletionRoutine** to register an *IoCompletion* routine, pass the IRP on to lower drivers, and return control from its own *DispatchDeviceControl* routine.

Almost all system-defined I/O control codes require the underlying device driver to transfer only modest amounts of data, usually much less than a PAGE_SIZE amount. As a general rule, higher-level drivers should handle these requests synchronously, as shown in the preceding code fragment, because the lower drivers return control so quickly. That is, the overhead of calling the higher-level driver's *IoCompletion* routine does not compensate for whatever additional IRP processing that driver can get done in such a short interval.

A higher-level driver that allocates IRPs with **IoBuildDeviceIoControlRequest** for an underlying device driver can handle these device control requests synchronously. The higher-level driver can wait for an optional event object to be passed to **IoBuildDeviceIoControlRequest** and associated with the driver-allocated IRP.

Dispatch(Internal)DeviceControl in Class/Port Drivers

6/25/2019 • 2 minutes to read • Edit Online

The higher-level driver of a class/port pair can sometimes complete IRPs in its *DispatchDeviceControl* routine. For example a class driver could, during initialization, gather and store information about the features of the underlying device, which might be sought in a subsequent **IRP_MJ_DEVICE_CONTROL** request, and thus save processing time by satisfying the request without passing it on to the underlying device driver. A class driver might also be designed to check the IRP's parameters and send only requests with valid parameters to the port driver.

Closely coupled class/port drivers also can define a set of driver-specific or device-specific internal I/O control codes that the class driver can use for **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests to the port driver.

For example, the *DispatchCreateClose* routines in the system keyboard and mouse class drivers send system-defined internal device control requests to enable or disable keyboard and mouse interrupts to the underlying port drivers. These system class drivers set up **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests for an underlying port driver. Any new keyboard or mouse port driver that interoperates with these system class drivers also must support these public internal device control requests.

The system parallel class/port driver model has similar features. New parallel class drivers can get support from the system parallel port driver by setting up IRPs for **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests with public IOCTL_PARALLEL_PORT_*XXX* control codes. You can replace the system parallel port driver, but any new driver also must support this set of public internal device control requests.

For more information about these public internal device control requests, see device-specific documentation in the Windows Driver Kit (WDK). For information about how to define private I/O control codes, see Using I/O Control Codes.

For a closely coupled pair of port/class drivers, the class driver might handle the processing of certain device control requests without passing them on to the port driver. In a new class/port driver pair, the class driver's *DispatchDeviceControl* routine can do either of the following:

- Check the validity of the parameters in its own I/O stack location, set the I/O status block if it finds any parameter errors, and call **IoCompleteRequest** with a *PriorityBoost* of IO_NO_INCREMENT; otherwise, call **IoGetNextIrpStackLocation** copy its own I/O stack location into the port driver's, and pass the IRP on with **IoCallDriver**.
- Or, do nothing more than set up the port driver's I/O stack location in the IRP without checking parameters and pass it on to the port driver for processing.

SCSI class drivers have special requirements for handling device control requests. For more information about these requirements, see Storage Drivers.

Guidelines for Writing Dispatch(Internal)DeviceControl Routines

6/25/2019 • 2 minutes to read • Edit Online

Keep the following points in mind when writing a *DispatchDeviceControl* or *DispatchInternalDeviceControl* routine:

At a minimum, a higher-level driver must copy the parameters for an **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL** request from its own I/O stack location in the IRP to the next-lower-level driver's I/O stack location. Then, it must call **IoCallDriver** with a pointer to the next-lower driver's device object and the IRP.

The higher-level driver should propagate the status value returned by **IoCallDriver** or set it in the returned IRP's I/O status block when it returns control for a request that lower drivers handle synchronously.

The underlying device driver must process device control requests unless it has a closely coupled class driver that completes a subset of these requests on its behalf. A device driver's *DispatchDeviceControl* routine usually begins processing these requests by turning on the **Parameters.DeviceIoControl.IoControlCode** in its I/O stack location of each IRP.

A lower-level device driver should check the parameters passed in with the request and fail the IRP with an appropriate error if any parameter is invalid. The most common check on the validity of parameters to these requests has the form:

```
    if (Irp->Parameters.DeviceIoControl.InputBufferLength <
            (sizeof(IOCTL_SPECIFIC_STRUCTURE))) {
        status = STATUS_XXX;
```

or

```
    if (Irp->Parameters.DeviceIoControl.OutputBufferLength <
            (sizeof(IOCTL_SPECIFIC_STRUCTURE))) {
        status = STATUS_XXX;
```

where the status value set is one of STATUS_BUFFER_TOO_SMALL or STATUS_INVALID_PARAMETER. Every device driver's *DispatchDeviceControl* or *DispatchInternalDeviceControl* routine must handle the receipt of an unrecognized I/O control code by setting the I/O status block with an appropriate NTSTATUS value, setting its **Information** field to zero, and completing the IRP with a *PriorityBoost* of IO_NO_INCREMENT.

The particular I/O control codes a device driver handles must include any device-type-specific, system-defined I/O control codes for the same type of device. See the device-specific sections of the Windows Driver Kit (WDK) for more information about the system requirements for each type of device and the corresponding (Windows SDK) header files, each beginning with the prefix *ntdd*, for declarations of the system-defined structures for these I/O control codes.

The class driver of a closely coupled class/port driver pair can process and complete a subset of device control requests without passing them on to the underlying port driver. However, such a class driver must pass on all valid device control requests that require a change of state for the device and those that require the return of volatile information about the device, such as its current baud rate, volume, or video mode.

DispatchPnP Routines

6/25/2019 • 4 minutes to read • Edit Online

A driver's *DispatchPnP* routine supports Plug and Play by handling IRPs for the **IRP_MJ_PNP** I/O function code. Associated with the **IRP_MJ_PNP** function code are several minor I/O function codes (see Plug and Play Minor IRPs), some of which all drivers must handle and some of which can be optionally handled. The PnP manager uses these minor function codes to direct drivers to start, stop, and remove devices and to query drivers about their devices.

All drivers for a device must have the opportunity to handle PnP IRPs for the device, except in a few cases where a function or filter driver is allowed to fail the IRP.

Each driver's *DispatchPnP* routine must follow these rules:

- A function or filter driver must pass PnP IRPs down to the next driver in the device stack, unless the function or filter driver handles the IRP and encounters a failure (due to insufficient resources, for example).

  All drivers for a device must have the opportunity to handle PnP IRPs for the device unless one of the drivers encounters an error. The PnP manager sends IRPs to the top driver in a device stack. Function and filter drivers pass the IRP down to the next driver, and the parent bus driver completes the IRP. See Passing PnP IRPs Down the Device Stack for more information.

  A driver can fail an IRP if it tries to handle the IRP and encounters an error (such as insufficient resources). If a driver receives an IRP with a code it does not handle, the driver must not fail the IRP. It must pass such an IRP down to the next driver without modifying the IRP's status.

- A driver must handle certain PnP IRPs and may optionally handle others.

  Each PnP driver is required to handle certain IRPs, such as **IRP_MN_REMOVE_DEVICE**, and can optionally handle others. See Plug and Play Minor IRPs for information about which IRPs are required and optional for each kind of driver (function drivers, filter drivers, and bus drivers).

  A driver can fail a required PnP IRP with an appropriate error status, but a driver must not return STATUS_NOT_SUPPORTED for such an IRP.

- If a driver handles a PnP IRP successfully, the driver sets the IRP status to success. It does not depend on another driver in the stack to set the status.

  A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS to inform the PnP manager that the driver handled the IRP successfully. For some IRPs, a non-bus driver might be able to rely on its parent bus driver to set the status to success. However, this is a risky practice. For consistency and robustness, a driver must set the IRP status to success for each PnP IRP it handles successfully.

- If a driver fails an IRP, the driver completes the IRP with an error status and does not pass the IRP down to the next driver.

  To fail an IRP like **IRP_MN_QUERY_STOP_DEVICE**, a driver sets **Irp->IoStatus.Status** to STATUS_UNSUCCESSFUL. Additional error status values for other IRPs include STATUS_INSUFFICIENT_RESOURCES and STATUS_INVALID_DEVICE_STATE.

  Drivers do not set STATUS_NOT_SUPPORTED for IRPs that they handle. This is the initial status set by the PnP manager. If an IRP is completed with this status, it means that no drivers in the stack handled the IRP; all drivers just passed the IRP to the next driver.

- A driver must handle a PnP IRP in its dispatch routine (on the IRP's way down the device stack), in an

IoCompletion routine (on the IRP's way back up the device stack), or both, as specified in the reference page for the IRP.

Some PnP IRPs, such as **IRP_MN_REMOVE_DEVICE**, must be handled first by the driver at the top of the device stack and then by each next-lower driver. Others, such as **IRP_MN_START_DEVICE**, must be handled first by the parent bus driver. Still others, such as **IRP_MN_QUERY_CAPABILITIES**, can be handled both on the way down the device stack and the way back up. See Plug and Play Minor IRPs for the rules that apply to each PnP IRP. See Postponing PnP IRP Processing Until Lower Drivers Finish For information about handling PnP IRPs that must be processed first by the parent bus driver.

- A driver must add information to an IRP on the IRP's way down the device stack and modify or remove information on the IRP's way back up.

  When returning information in response to a PnP query IRP, a driver must follow this convention to enable orderly information passing by the layered drivers for a device.

- Except where explicitly documented, a driver must not depend on PnP IRPs being sent in any particular order.

- When a driver sends a PnP IRP, it must send the IRP to the top driver in the device stack.

  Most PnP IRPs are sent by the PnP manager, but some can be sent by drivers (for example, **IRP_MN_QUERY_INTERFACE**). A driver must send a PnP IRP to the driver at the top of the device stack. Call **IoGetAttachedDeviceReference** to get a pointer to the device object for the driver at the top of the device stack.

You should test your drivers with a checked build of the operating system. The checked build of the system verifies whether a driver follows many of the PnP rules listed above.

DispatchPower Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *DispatchPower* routine supports power management by handling IRPs for the **IRP_MJ_POWER** I/O function code. Associated with the **IRP_MJ_POWER** function code are several minor I/O function codes for Power Management. The power manager uses these minor function codes to direct drivers to change power states, to wait for and respond to system wake-up events, and to query drivers about their devices.

Each driver's *DispatchPower* routine performs the following tasks:

- Handle the IRP if possible.
- Pass the IRP to the next lower driver in the device stack, using **PoCallDriver**.
- If a bus driver, perform the requested power operation on the device and complete the IRP.

All drivers for a device must have the opportunity to handle power IRPs for the device, except in a few cases where a function or filter driver is allowed to fail the IRP. Most function and filter drivers either perform some processing or set an *IoCompletion* routine for each power IRP, then pass the IRP down to the next lower driver without completing it. Eventually the IRP reaches the bus driver, which physically changes the power state of the device if required and completes the IRP.

When the IRP has been completed, the I/O manager calls any *IoCompletion* routines set by drivers as the IRP traveled down the device stack. Whether a driver needs to set a completion routine depends upon the type of IRP and the driver's individual requirements.

Power IRPs that power up a device must be handled first by the lowest driver in the device stack (the underlying bus driver) and then by each successive driver up the stack. Power IRPs that power down a device must be handled first by the driver at the top of the device stack and then by each successive driver going down the stack.

Special Handling for Removable Devices

In their *DispatchPower* routines, drivers of removable devices should check to see whether the device is still present. If the device has been removed, the driver should not pass the IRP down to the next lower driver. Instead, the driver should do the following:

- Call **PoStartNextPowerIrp** to begin processing the next power IRP.
- Set **Irp->IoStatus.Status** to STATUS_DELETE_PENDING.
- Call **IoCompleteRequest**, specifying IO_NO_INCREMENT, to complete the IRP.
- Return STATUS_DELETE_PENDING.

DispatchQueryInformation Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *DispatchQueryInformation* routine handles IRPs for the **IRP_MJ_QUERY_INFORMATION** I/O function code. Driver support for this I/O function code is optional, and typically appears in higher-level or file system drivers. This request is sent by the I/O manager and other operating system components, as well as other kernel-mode drivers. For example, it is sent when a user-mode application calls **GetFileInformationByHandle**, and when a kernel-mode component calls **ZwQueryInformationFile**.

DispatchSetInformation Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *DispatchSetInformation* routine handles IRPs for the **IRP_MJ_SET_INFORMATION** I/O function code. Driver support for this I/O function code is optional, and typically appears in higher-level or file system drivers. This request is sent by the I/O manager and other operating system components, as well as other kernel-mode drivers. For example, it is sent when a user-mode application calls **SetEndOfFile**, and when a kernel-mode component calls **ZwSetInformationFile**.

DispatchFlushBuffers Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *DispatchFlushBuffers* routine handles IRPs for the **IRP_MJ_FLUSH_BUFFERS** I/O function code. Driver support for this I/O function code is optional, but all file system and filter drivers that maintain internal data buffers must handle it to preserve changes to file data or metadata across system shutdowns. This request is sent by the I/O manager and other operating system components, as well as other kernel-mode drivers, when buffered data needs to be flushed to disk. For example, it is sent when a user-mode application calls **FlushFileBuffers**.

DispatchShutdown Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *DispatchShutdown* routine handles IRPs for the **IRP_MJ_SHUTDOWN** I/O function code. Drivers of mass-storage devices that have internal caches for data must handle this request. Drivers of mass-storage devices and intermediate drivers layered over them also must handle this request if an underlying driver maintains internal buffers for data.

DispatchSystemControl Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *DispatchSystemControl* routine handles IRPs for the **IRP_MJ_SYSTEM_CONTROL** I/O function code.

All drivers must provide a *DispatchSystemControl* routine. The purpose of this routine is to provide support for Windows Management Instrumentation (WMI). Regardless of whether a driver supports WMI, this routine must pass the IRP to the next-lower driver.

To learn how to implement a *DispatchSystemControl* routine, and how to support WMI in general, see Windows Management Instrumentation.

Writing an Unload Routine

6/25/2019 • 2 minutes to read • Edit Online

Any driver that can be replaced, or unloaded and reloaded, while the system is running must have an *Unload* routine. All WDM drivers must have *Unload* routines.

Although *Unload* routines are optional for non-WDM drivers, Driver Verifier will fail any driver that does not provide an *Unload* routine.

This section contains the following topics:

Unload Routine Environment

6/25/2019 • 2 minutes to read • Edit Online

The operating system unloads a driver when the driver is being replaced or when all of the devices that the driver services have been removed. The PnP manager calls a PnP driver's *Unload* routine if the driver has no more device objects after it handles an **IRP_MN_REMOVE_DEVICE** request.

At the start of the unloading sequence, the I/O manager or PnP manager marks the driver object and its device objects as "Unload Pending". After a driver has been marked as "Unload Pending", no additional drivers can attach to that driver, nor can any additional references be made to the driver's device objects. The driver can complete outstanding IRPs, but the system will not send any new IRPs to the driver.

The I/O manager calls a driver's *Unload* routine when all of the following are true:

- No references remain to any of the device objects the driver has created. In other words, no files associated with the underlying device can be open, nor can any IRPs be outstanding for any of the driver's device objects.
- No other drivers remain attached to this driver.
- The driver has called **IoUnregisterPlugPlayNotification** to unregister all PnP notifications for which it previously registered.

Note that the *Unload* routine is not called if a driver's **DriverEntry** routine returns a failure status. In this case, the I/O manager simply frees the memory space taken up by the driver.

Neither the PnP manager nor the I/O manager calls *Unload* routines at system shutdown time. A driver that must perform shutdown processing should register a *DispatchShutdown* routine.

Unload Routine Functionality

6/25/2019 • 2 minutes to read • Edit Online

The responsibilities of a driver's *Unload* routine depend on whether the driver supports PnP or not.

Just as the **DriverEntry** routines of PnP drivers are usually simple, so are their *Unload* routines, as described in A PnP Driver's Unload Routine.

A non-PnP driver's *Unload* routine must free device objects and release driver-allocated resources. In short, it must undo the work performed by its corresponding **DriverEntry** and *Reinitialize* routines in initializing the driver, its devices, and its resources. See A Non-PnP Driver's Unload Routine for details.

PnP Driver's Unload Routine

6/25/2019 • 2 minutes to read • Edit Online

A PnP driver must have an *Unload* routine that removes any driver-specific resources, such as memory, threads, and events, that are created by the **DriverEntry** routine. If there are no driver-specific resources to remove, the driver must still have an *Unload* routine but it can simply return.

A driver's *Unload* routine can be called at any time after all the driver's devices have been removed. The PnP manager calls a driver's *Unload* routine in the context of a system thread at IRQL = PASSIVE_LEVEL.

PnP drivers free device-specific resources and device objects in response to PnP device-removal IRPs. The PnP manager sends these IRPs on behalf of each PnP device it enumerates as well as any root-enumerated legacy devices a driver reports using **IoReportDetectedDevice**.

Consequently, the *Unload* routines of PnP drivers are usually simple, often consisting only of a **return** statement. However, if the driver allocated any driver-wide resources in its **DriverEntry** routine, it must deallocate those resources in its *Unload* routine unless it has already done so. In general, the process of unloading a PnP driver is a synchronous operation.

The I/O manager frees the driver object and any driver object extension that the driver allocated using **IoAllocateDriverObjectExtension**.

Non-PnP Driver's Unload Routine

6/25/2019 • 2 minutes to read • Edit Online

Earlier drivers and high-level file system drivers, which do not handle PnP device-removal requests, must release resources, delete device objects, and detach from the device stack in their *Unload* routines.

If it has not done so already, the first thing a legacy device driver should do in its *Unload* routine is to disable interrupts from the device. Otherwise, its ISR might be called to handle a device interrupt while the *Unload* routine is releasing resources in the device extension that the ISR needs to handle the interrupt. Even if its ISR runs successfully in these circumstances, the *DpcForIsr* or *CustomDpc* routine that the ISR queues, and possibly other driver routines that run at IRQL >= DISPATCH_LEVEL, will execute before the *Unload* routine regains control, thereby increasing the likelihood that the *Unload* routine has deleted a resource that another driver routine references. See Managing Hardware Priorities for more information.

After disabling interrupts, file system and legacy drivers must release resources and objects. For details, see the following two sections:

Releasing Driver-Allocated Resources

Releasing Device and Controller Objects

Releasing Driver-Allocated Resources

6/25/2019 • 3 minutes to read • Edit Online

The specifics of how a driver uses the registry, sets up system objects and resources in its device extensions, controller extension, or driver-allocated nonpaged pool varies from driver to driver. However, any *Unload* routine must release the resources a driver is using in stages.

Any driver's *Unload* routine must ensure that no other driver routine is currently using or might shortly be using a particular resource before it releases that resource.

In general, an *Unload* routine releases all driver-allocated resources in the following stages:

1. If the driver has not already done so, disable interrupts on any physical devices, if possible, and then call **IoDisconnectInterrupt** as soon as interrupts are disabled.

2. Ensure that no other driver routine can reference the resources that the *Unload* routine intends to release.

   For example, an *Unload* routine must call **IoStopTimer** if the driver's *IoTimer* routine is currently enabled for a particular device object. It must ensure that no thread is waiting for any of the driver's dispatcher objects and that its timer objects are not queued for calls to its *CustomTimerDpc* routines before it frees the storage for its dispatcher objects. It must call **KeRemoveQueueDpc** if it has a *CustomDpc* routine that the ISR might have queued, and so on.

   If the driver called **IoQueueWorkItem**, it must ensure that the work item has completed. **IoQueueWorkItem** takes out a reference on the associated device object; the driver cannot be unloaded if any such references remain.

   If the driver called **PsCreateSystemThread**, the *Unload* routine also must cause the driver-created thread to be run so that the thread itself can call **PsTerminateSystemThread** before the driver is unloaded. A driver cannot release a driver-created system thread by calling **ZwClose** with the *ThreadHandle* returned by **PsCreateSystemThread**.

3. Release any device-specific resources that the driver allocated. Doing so might involve calling the following system support routines:

   - **IoDeleteSymbolicLink** if the **DriverEntry** or *Reinitialize* routine called **IoCreateSymbolicLink** or **IoCreateUnprotectedSymbolicLink**, and **IoDeassignArcName** if the driver called **IoAssignArcName**.
   - **ExFreePool** if **DriverEntry** or any other driver routine called **ExAllocatePoolWithTag** and the driver has not yet released the allocated memory.
   - **MmUnmapIoSpace** if the **DriverEntry** or *Reinitialize* routine called **MmMapIoSpace**.
   - **MmFreeNonCachedMemory** if the **DriverEntry** or *Reinitialize* routine called **MmAllocateNonCachedMemory**.
   - **MmFreeContiguousMemory** if the **DriverEntry** or *Reinitialize* routine called **MmAllocateContiguousMemory**.
   - **FreeCommonBuffer** if the **DriverEntry** or *Reinitialize* routine called **AllocateCommonBuffer**.
   - **IoAssignResources** or **IoReportResourceUsage** if the **DriverEntry** or *Reinitialize* routine called one of these support routines or **HalAssignSlotResources** to claim hardware resources in the configuration registry for itself and/or for its physical devices individually.

4. Release system objects and resources that the **DriverEntry** or *Reinitialize* routine set up in the device extension of the device objects or in the controller extension of the controller object (if it created one). In particular, the driver must do the following before it attempts to delete the device object (**IoDeleteDevice**) or controller object (**IoDeleteController**):

    - Call **IoDisconnectInterrupt** to free the interrupt object pointer stored in the corresponding device or controller extension.
    - Call **ObDereferenceObject** with the pointer to the next-lower driver's file object if it called **IoGetDeviceObjectPointer** and stored this pointer in a device or controller extension.
    - Call **IoDetachDevice** with the pointer to the lower driver's device object if it called **IoAttachDevice** or **IoAttachDeviceToDeviceStack** and stored this pointer in a device or controller extension.
5. Free the hardware resources that the **DriverEntry** or *Reinitialize* routine claimed for the driver's physical devices, if any, in the registry under the **\Registry\Machine\Hardware\ResourceMap** tree.

6. Remove any names for its devices that the **DriverEntry** or *Reinitialize* routine stored in the registry under the **\Registry..\DeviceMap** tree, as well.

After the driver has released device, system, and hardware resources, it can delete its device and controller objects, as described in the following section.

Releasing Device and Controller Objects

6/25/2019 • 2 minutes to read • Edit Online

Before a driver deletes a device or controller object, it must release its references to external resources, such as pointers to other drivers' objects and/or to interrupt objects, that it stored in the corresponding device or controller extension. It can then call **IoDeleteDevice** for each device object that the driver created. A non-WDM driver that previously called **IoCreateController** must also call **IoDeleteController**.

Any Kernel-defined object for which the driver provides storage in a device extension is automatically freed when the *Unload* routine calls **IoDeleteDevice** with the corresponding device object. In general, any object that the **DriverEntry** or *Reinitialize* routine set up by calling **KeInitialize*Xxx*** can be freed by a call to **IoDeleteDevice** if the driver provided storage for that object in its device extension. For example, if a driver has a *CustomTimerDpc* routine and has provided storage for the necessary DPC and timer objects in its device extension, the call to **IoDeleteDevice** releases these system resources.

Similarly, any Kernel-defined object for which the driver provides storage in a controller extension is automatically freed when the *Unload* routine calls **IoDeleteController** with the corresponding controller object.

If the **DriverEntry** or *Reinitialize* routine called **IoGetConfigurationInformation** to increment the count for a particular type of device, the *Unload* routine also must call **IoGetConfigurationInformation** and decrement the count for its devices in the I/O manager's global configuration information structure as it deletes the corresponding device objects.

Before it returns control, an *Unload* routine also is responsible for freeing any other driver-allocated resources that have not yet been freed by other driver routines.

Introduction to Device Objects

6/25/2019 • 2 minutes to read • Edit Online

The operating system represents devices by *device objects*. One or more device objects are associated with each device. Device objects serve as the target of all operations on the device.

Kernel-mode drivers must create at least one device object for each device, with the following exceptions:

- Minidrivers that have an associated class or port driver do not have to create their own device objects. The class or port driver creates the device objects, and dispatches operations to the minidriver.
- Drivers that are part of device type-specific subsystems, such as NDIS miniport drivers, have their device objects created by the subsystem.

See the documentation for your particular device type to determine if your driver creates its own device objects.

Some device objects do not represent physical devices. A software-only driver, which handles I/O requests but does not pass those requests to hardware, still must create a device object to represent the target of its operations.

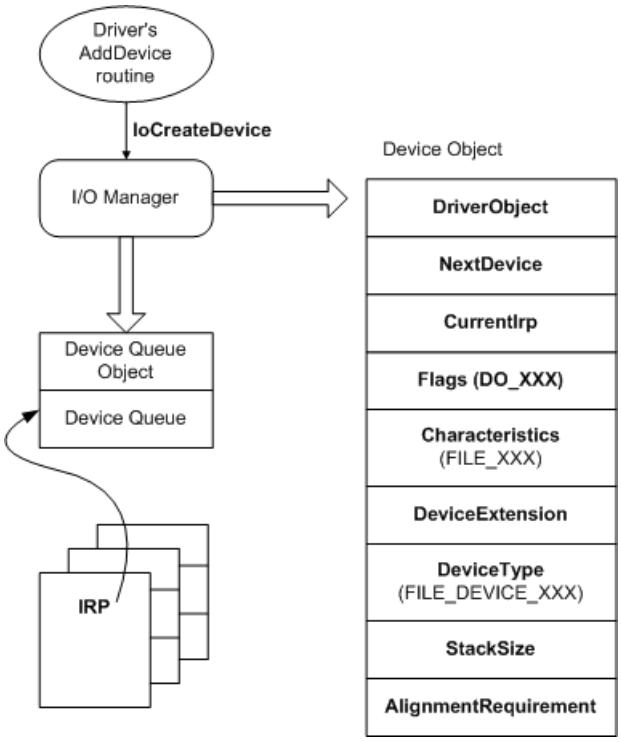
For more information about how your driver can create device objects, see Creating a Device Object.

Devices are usually represented by multiple device objects, one for each driver in the driver stack that handles I/O requests for the device. The device objects for a device are organized into a *device stack*. Whenever an operation is performed on a device, the system passes an **IRP** data structure to the driver for the top device object in the device stack. Each driver either handles the IRP or passes it to the driver that is associated with the next-lower device object in the device stack. For more information about device stacks, see WDM Device Objects and Device Stacks. For more information about IRPs, see Handling IRPs.

Device objects are represented by **DEVICE_OBJECT** structures, which are managed by the object manager. The object manager provides the same capabilities for device objects that it does for other system objects. In particular, a device object can be named, and a named device object can have handles opened on it. For more information about named device objects, see Named Device Objects.

The system provides dedicated storage for each device object, called the device extension, which the driver can use for device-specific storage. The device extension is created and freed by the system along with the device object. For more information, see Device Extensions.

The following figure illustrates the relationship between device objects and the I/O manager.



The figure shows the members of the **DEVICE_OBJECT** structure that are of interest to a driver writer. For more information about these members, see Creating a Device Object, Initializing a Device Object, and Properties of Device Objects.

Types of WDM Device Objects

6/25/2019 • 2 minutes to read • Edit Online

There are three kinds of WDM device objects:

1. Physical Device Object (PDO) – represents a device on a bus to a bus driver.
2. Functional Device Object (FDO) – represents a device to a function driver.
3. Filter Device Object (filter DO) – represents a device to a filter driver.

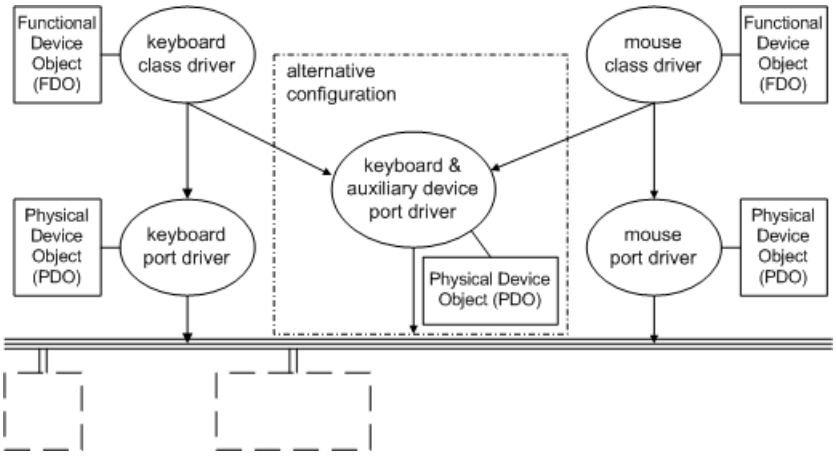
The three kinds of device objects are all of the type **DEVICE_OBJECT**, but are used differently and can have different device extensions.

A driver adds itself to the stack of drivers that handle I/O for a device by creating a device object (**IoCreateDevice**) and attaching it to the device stack (**IoAttachDeviceToDeviceStack**). **IoAttachDeviceToDeviceStack** determines the current top of the device stack and attaches the new device object to the top of the device stack.

Example WDM Device Objects

6/25/2019 • 2 minutes to read • Edit Online

The following figure illustrates the device objects that represent the keyboard and mouse devices shown previously in the figure illustrating Keyboard and Mouse Hardware Configurations. The keyboard and mouse drivers shown in the figure illustrating Keyboard and Mouse Driver Layers create these device objects by calling an I/O support routine (**IoCreateDevice**).



For the keyboard and mouse devices, both their respective port and class drivers create device objects. The port driver creates a physical device object (PDO) to represent the physical port. Each class driver creates its own functional device object (FDO) to represent the keyboard or mouse device as a target for I/O requests.

Each class driver calls an I/O support routine to get a pointer to the next-lower-level driver's device object, so the class driver can chain itself above that driver, which is the port driver. Then the class driver can send I/O requests down to the port driver for the target PDO representing its physical device.

An optional filter driver added to the configuration would create a filter device object (filter DO). Like the class driver, an optional filter driver chains itself to the next-lower driver in the device stack and sends I/O requests for the target PDO down to the next-lower driver.

As shown previously in the Keyboard and Mouse Driver Layers figure, each port driver is a bus (lowest-level) driver, so every port driver of a device that generates interrupts must set up interrupt object(s) and register an ISR.

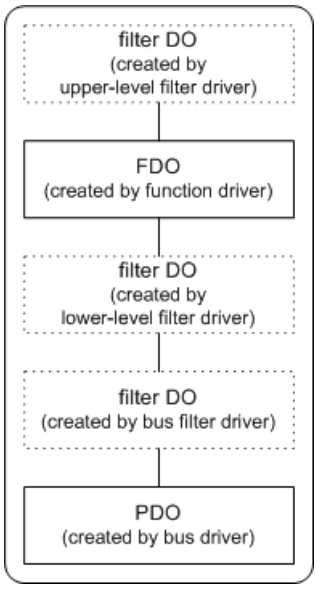
A dual-device port driver, like the i8042 driver for the keyboard and auxiliary device controller shown in the Keyboard and Mouse Hardware Configurations if each device uses a different interrupt vector. When writing such a driver, you can either implement separate ISRs for each device or implement a single ISR for both devices.

When Are WDM Device Objects Created?

6/25/2019 • 3 minutes to read • Edit Online

This section describes each kind of device object and mentions when each is created.

The following figure shows the possible kinds of device objects that can be attached in a device stack, representing the drivers handling I/O requests for a device.



Starting at the bottom of this figure:

- A bus driver creates a PDO for each device that it enumerates on its bus.

  A bus driver creates a PDO for a child device when it enumerates the device. A bus driver enumerates a device in response to an **IRP_MN_QUERY_DEVICE_RELATIONS** request for **BusRelations** from the PnP manager. The bus driver creates a PDO for a child device if the device has been added to the bus since the last time the bus driver responded to a query-relations request for **BusRelations** (or if this is the first query-relations request since the machine was booted).

  A PDO represents the device to the bus driver, as well as to other kernel-mode system components such as the power manager, the PnP manager, and the I/O manager.

  Other drivers for a device attach device objects on top of the PDO, but the PDO is always at the bottom of the device stack.

- Optional bus filter drivers create filter DOs for each device they filter.

  When the PnP manager detects a new device in a **BusRelations** list, it determines whether there are any bus filter drivers for the device. If so, for each such driver the PnP manager ensures it is loaded (calls **DriverEntry** if necessary) and calls the driver's *AddDevice* routine. If the bus filter driver filters operations for this device, the filter driver creates a device object and attaches it to the device stack in its *AddDevice* routine. If more than one bus filter driver exists and is relevant to this device, each such filter driver creates and attaches its own device object.

- Optional, lower-level filter drivers create filter DOs for each device they filter.

  If an optional, lower-level filter driver exists for this device, the PnP manager ensures that such a driver is loaded after the bus driver and any bus filter drivers. The PnP manager calls the filter driver's *AddDevice* routine. In its *AddDevice* routine, the lower-level filter driver creates a filter DO for the device and attaches it

to the device stack. If more than one lower-level filter driver exists, each such driver would create and attach its own filter DO.

- The function driver creates an FDO for the device.

  The PnP manager ensures that the function driver for the device is loaded and calls the function driver's *AddDevice* routine. The function driver creates an FDO and attaches it to the device stack.

- Optional, upper-level filter drivers create a filter DO for each device they filter.

  If any optional, upper-level filter drivers exist for the device, the PnP manager ensures they are loaded after the function driver and calls their *AddDevice* routines. Each such filter driver attaches its device object to the device stack.

In summary, the device stack contains a device object for each driver that is involved in handling I/O to a particular device. The parent bus driver has a PDO, the function driver has an FDO, and each optional filter driver has a filter DO.

Note that all devices, bus adapter/controller devices and nonbus devices, have a PDO and an FDO in their device stack. The PDO for a bus adapter/controller is created by the bus driver for the parent bus. For example, if a SCSI adapter plugs into a PCI bus, the PCI bus driver creates a PDO for the SCSI adapter.

If a device is being used in raw mode, there are no function or filter drivers (no FDO or filter DOs). There is just a PDO for the parent bus driver and zero or more bus filter DOs.

See Creating a Device Object for information about which driver routines are responsible for creating and attaching device objects.

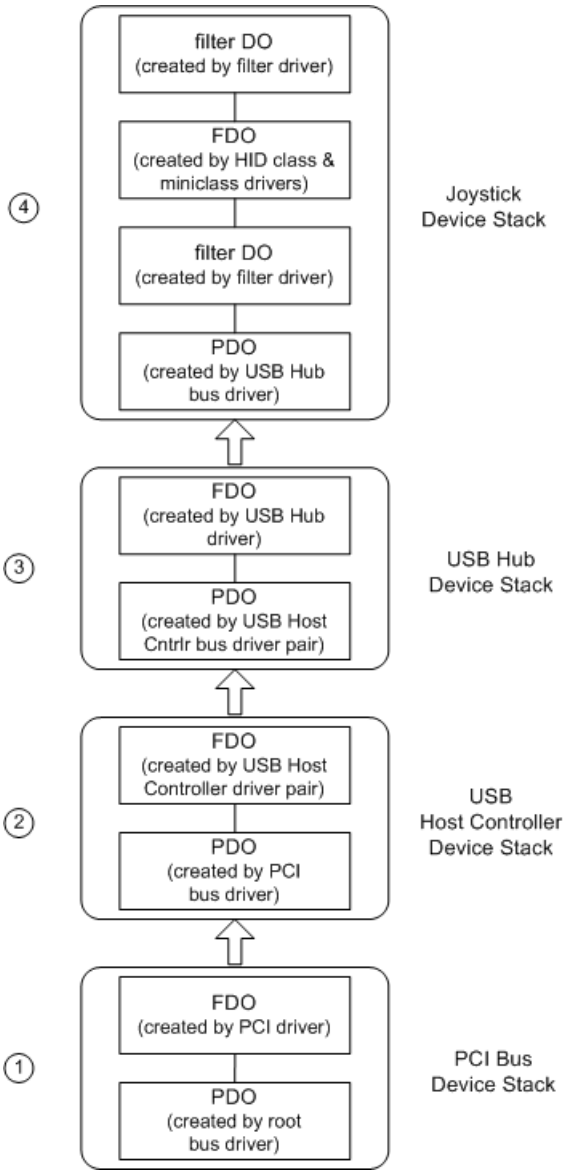
The device stack plus some additional information constitutes the *devnode* for a device. The PnP manager maintains information in a device's devnode such as whether the device has been started and which drivers, if any, have registered for notification of changes on the device. The kernel debugger command **!devnode** displays information about a devnode.

Example WDM Device Stack

6/25/2019 • 3 minutes to read • Edit Online

This section describes the device objects created by a possible set of drivers for USB hardware to illustrate WDM device objects and how they are layered.

The following figure shows the device objects that are created by the sample drivers described in WDM Driver Layers: An Example.



Starting at the bottom of this figure, the device objects in the sample device stacks include:

1. A PDO and an FDO for the PCI bus.

   The root bus driver enumerates the internal system bus (the root bus) and creates a PDO for each device it finds. One of these PDOs is for the PCI bus. (The PDO and FDO for the root bus are not shown in the figure.)

   The PnP manager identifies the PCI driver as the function driver for the PCI bus, loads the driver (if it is not already loaded), and passes the PDO to the PCI driver. In its *AddDevice* routine, the PCI driver creates an FDO for the PCI bus (**IoCreateDevice**) and attaches the FDO to the device stack (**IoAttachDeviceToDeviceStack**) for the PCI bus. The PCI driver creates and attaches this FDO as part of

its responsibilities as the function driver for the PCI bus.

There are no filter drivers for the PCI bus in this example.

2. A PDO and an FDO for the USB host controller.

   The PnP manager directs the PCI driver to start its device (**IRP_MN_START_DEVICE**) and then queries the PCI driver for its children (**IRP_MN_QUERY_DEVICE_RELATIONS** with relation type of **BusRelations**). In response, the PCI driver enumerates the devices on its bus. In this example, the PCI driver finds a USB host controller and creates a PDO for that device. The wide arrow in the figure indicates that the USB host controller is a "child" of the PCI bus. The PCI driver creates PDOs for its child devices as part of its responsibilities as the bus driver for the PCI bus.

   The PnP manager identifies the USB host controller miniclass/class driver pair as the function driver for the USB host controller and loads the driver pair. The PnP manager calls the driver pair at the appropriate time to create and attach an FDO for the USB host controller.

   There are no filter drivers for the USB host controller in this example.

3. A PDO and an FDO for the USB hub.

   The USB host controller enumerates its bus, locates the USB hub in the sole port, and creates a PDO for the hub. The USB hub driver creates and attaches an FDO for the hub.

   There are no filter drivers for the USB hub in this example.

4. A PDO, an FDO, and two filter DOs for the joystick device.

   The USB hub driver enumerates its bus, locates a HID device (the joystick), and creates a PDO for the joystick.

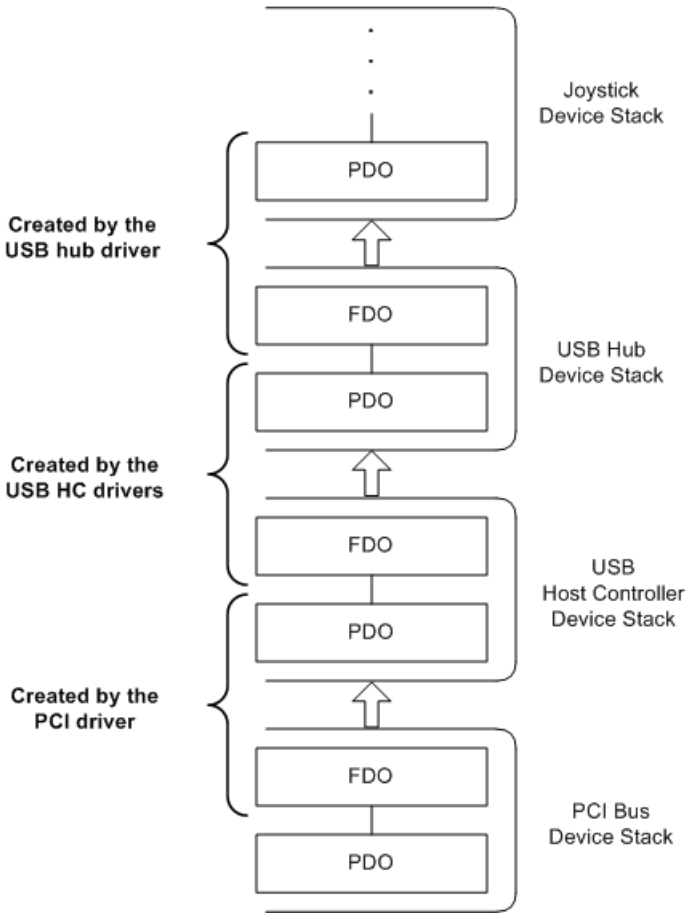
   In this example, a lower-level filter driver has been set up in the registry for joystick devices, so the PnP manager loads the filter driver. The filter driver determines that it is relevant to the device and creates and attaches a filter DO to the device stack.

   The PnP manager determines that the function driver for the joystick device is the HID class/miniclass driver pair and loads those drivers. The driver pair consists of a miniclass driver linked to a class driver DLL; together they act as one function driver for the device. The class/miniclass driver pair creates one device object, the FDO, and attaches it to the device stack.

   An upper-level filter driver creates and attaches a filter DO to the device stack, in a manner similar to the lower-level filter.

Note that the PDO created by the parent bus driver is always at the bottom of the device stack for a particular device. When drivers handle PnP or power IRPs, they must pass each IRP all the way down the device stack to the PDO and its associated bus driver.

The following figure shows the same device stacks as the previous figure, but emphasizes which device objects are created and managed by which drivers.



A bus driver spans more than one device stack. A bus driver creates the FDO for its bus adapter/controller and creates a PDO for each of its child devices.

Creating a Device Object

6/25/2019 • 3 minutes to read • Edit Online

A monolithic driver must create a device object for each physical, logical, or virtual device for which it handles I/O requests. A driver that does not create a device object for a device does not receive any IRPs for the device.

In some technology areas, a minidriver that is associated with a class driver or port driver does not have to create its own device objects. Instead, the class or port driver creates the device object, and receives all IRPs for the device. The class or port driver then uses a driver-specific method to pass the I/O request to the minidriver. See the documentation for your particular technology area to determine if Microsoft supplies a class or port driver that creates device objects on behalf of your driver.

Drivers call either **IoCreateDevice** or **IoCreateDeviceSecure** to create their device objects. For more information about which routine to use, see the following sections.

Creating Device Objects for WDM Function and Filter Drivers

Creating Device Objects for WDM Bus Drivers

Creating Device Objects for Non-WDM Drivers

When the driver creates a device object, it supplies the following information to **IoCreateDevice** or **IoCreateDeviceSecure**:

- The size of the device's *device extension*. The device extension is a system-allocated storage area that the driver can use for device-specific storage. For more information, see Device Extensions.
- A system-defined constant, indicating the **DeviceType** represented by the device object. For more information, see Specifying Device Types.
- One or more ORed, system-defined constants that indicate the device characteristics for the device. For more information, see Specifying Device Characteristics.
- A Boolean value, named *Exclusive*, that specifies whether a bit in the device object's **Flags** should be set with DO_EXCLUSIVE, indicating the driver services an exclusive device, such as a video, serial, parallel, or sound device. WDM drivers must set *Exclusive* to **FALSE**. For more information, see Specifying Exclusive Access to Device Objects.
- A pointer to the driver object for the driver. A WDM function or filter driver receives a pointer to its driver object as a parameter to its *AddDevice* routine. All drivers receive a pointer to their driver object in their **DriverEntry** routine. The system uses this pointer to associate the driver with its device object.
- An optional pointer to a null-terminated Unicode string (*DeviceName*) naming the device. WDM drivers, other than bus drivers, do not supply a device name; doing so bypasses the PnP manager's security features. For more information, see Named Device Objects.

If the call to **IoCreateDevice** or **IoCreateDeviceSecure** succeeds, the I/O manager provides storage for the device object itself and for all other data structures associated with the device object, including the device extension, which it initializes with zeros.

Creating Device Objects for WDM Function and Filter Drivers

WDM drivers, other than bus drivers, call **IoCreateDevice** to create their device objects. Most WDM drivers create their device objects from within their *AddDevice* routines. Some drivers, such as disk drivers that must respond to drive layout IOCTLs, call **IoCreateDevice** from a dispatch routine.

Unless device type-specific sections of the Windows Driver Kit (WDK) documentation state otherwise, your driver should create its device objects in its *AddDevice* routine. For more information, see Writing an AddDevice Routine.

Creating Device Objects for WDM Bus Drivers

A WDM bus driver creates a PDO when it is enumerating a new device in response to an **IRP_MN_QUERY_DEVICE_RELATIONS** request, if the relation type is **BusRelations**.

The following rules determine if a bus driver calls **IoCreateDevice** or **IoCreateDeviceSecure** to create a device object:

- If a device can be used in *raw mode*, then it must call **IoCreateDeviceSecure**.
- If the device is not raw-mode capable, then the bus driver can use either **IoCreateDevice** or **IoCreateDeviceSecure**. **IoCreateDevice** can be used when the default system security for devices on the bus is adequate; **IoCreateDeviceSecure** can be used to specify a more stringent security descriptor. For more information, see Controlling Device Access.

Creating Device Objects for Non-WDM Drivers

A non-WDM driver uses **IoCreateDevice** to create unnamed device objects, and **IoCreateDeviceSecure** to create named device objects. Note the unnamed device objects of a non-WDM driver are not accessible from user mode, so the driver usually must create at least one named object.

Initializing a Device Object

6/25/2019 • 3 minutes to read • Edit Online

After **IoCreateDevice** returns, giving the caller a pointer to a *DeviceObject* that contains a pointer to the *device extension*, drivers must set up certain fields in the device objects for their respective physical, logical, and/or virtual devices.

IoCreateDevice sets the **StackSize** field of a newly created device object to one. A lowest-level driver can ignore this field. When a higher-level driver calls **IoAttachDeviceToDeviceStack** to attach itself to the next-lower driver, that routine automatically sets the **StackSize** field in the device object to that of the next-lower driver's device object plus one. For some device types, however, the higher-level driver might need to set the **StackSize** field to a greater value, as noted in the device-specific documentation. Setting the stack size ensures that IRPs sent to the higher-level driver will contain a driver-specific I/O stack location, plus the correct number of I/O stack locations for all lower-level drivers in the chain.

IoCreateDevice sets the **AlignmentRequirement** field of a newly created device object to the processor's data cache line size minus one, to ensure that buffers used in direct I/O are aligned correctly. After **IoCreateDevice** returns, lowest-level physical device drivers must do the following:

1. Subtract one from the alignment requirement of the device.
2. Compare the result of step 1 with the current value of the device object's **AlignmentRequirement**.
3. If the device's alignment requirement is greater, set**AlignmentRequirement** to the result of step 1. Otherwise, leave the **AlignmentRequirement** value as set by **IoCreateDevice**.

After any higher-level driver chains itself over another driver by calling **IoGetDeviceObjectPointer**, the higher-level driver must set the **AlignmentRequirement** field of its newly created device object to that of the next-lower-level driver's device object. As a general rule, a higher-level driver should not change this value. If a higher-level driver calls **IoAttachDevice** or **IoAttachDeviceToDeviceStack**, those routines automatically set the **AlignmentRequirement** field in the device object to that of the lower-level driver's device object.

IoGetDeviceObjectPointer returns pointers both to the lower-level driver's device object and to the associated file object. Only an FSD (or, possibly, another highest-level driver) can use the returned file object pointer. An intermediate driver that calls **IoGetDeviceObjectPointer** should save this file object pointer so it can be dereferenced by calling **ObDereferenceObject** when the driver is unloaded.

After an FSD mounts the volume containing the file object that represents a lower driver's device object, an intermediate driver cannot chain itself between the file system and the lower driver by calling **IoAttachDevice** or **IoAttachDeviceToDeviceStack**. Additionally, an FSD can set the **SectorSize** member of the device object based on the geometry of the underlying volume hardware when a mount occurs. For more information, see **DEVICE_OBJECT**.

An intermediate or lowest-level driver also sets a bit in the device object's **Flags** by ORing it either with DO_DIRECT_IO or with DO_BUFFERED_IO in every device object it creates. Highest-level drivers of logical or virtual devices can avoid setting **Flags** for either buffered or direct I/O if the driver writer decides the additional work involved will pay off in better driver performance. An intermediate driver must set up the **Flags** field of its device object to match that of the next-lower driver's device object.

Setting up a device object **Flags** field with DO_DIRECT_IO or DO_BUFFERED_IO determines how the I/O manager passes access to user buffers in all data transfer requests subsequently sent to the driver.

The driver can then set any other device-dependent values in the device object. For example, non-WDM drivers

for removable-media devices must OR the device object's **Flags** member with DO_VERIFY_VOLUME if they detect (or suspect) a change in media during I/O operations. (See Supporting Removable Media for more information.) Drivers of devices that require inrush power must OR the **Flags** member with DO_POWER_INRUSH, and drivers of devices that are not on the system paging path must OR the **Flags** member with DO_POWER_PAGABLE. Function and filter drivers must clear the DO_DEVICE_INITIALIZING flag.

After initializing the device object, a driver can also initialize any Kernel-defined objects and other system-defined data structures for which it has provided storage in the device extension. Precisely when a driver performs these tasks depends on its device, the type of the object, and/or the nature of the data. In general, any objects or data structures that can persist through PnP start and stop requests can be initialized in the *AddDevice* routine. Those that require resource information provided with a PnP **IRP_MN_START_DEVICE** request, or that might require changes when the device is stopped and/or restarted, should be initialized when the driver handles the **IRP_MN_START_DEVICE** request. For more information about *AddDevice* routines, see Writing an AddDevice Routine.

Named Device Objects

6/25/2019 • 2 minutes to read • Edit Online

A device object, like all object manager objects, can be named or unnamed. When a user-mode application makes an I/O request, it specifies the target of the operation by name. The object manager resolves the name to determine the destination of the I/O request.

> **IMPORTANT**
>
> To help increase driver security name device objects only when necessary. Named device objects are generally only necessary for legacy reasons, for example if you have an application that expects to open the device using a particular name or if you're using a non-PNP device/control device. Note that WDF drivers do not need to name their PnP device in order to create a symbolic link using WdfDeviceCreateSymbolicLink.

A driver can specify a name for a device object when it calls **IoCreateDevice** or **IoCreateDeviceSecure** to create the device object. For more information about when and how to name a device object, see NT Device Names.

A named device object can also have an MS-DOS device name, which is a symbolic link created by **IoCreateSymbolicLink** or **IoCreateUnprotectedSymbolicLink**. WDM drivers do not in general require an MS-DOS device name. For more information, see MS-DOS Device Names.

> **IMPORTANT**
>
> If you use a named device object you can use IoCreateDeviceSecure and specify a SDDL to help secure it. When you implement IoCreateDeviceSecure always specify a custom class GUID for DeviceClassGuid. You should not specify an existing class GUID here. Doing so has the potential to break security settings or compatibility for other devices belonging to that class. For more information, see WdmlibIoCreateDeviceSecure.
>
> In order to allow applications or other WDF drivers to access your PnP device, you should use device interfaces. For more information, see Using Device Interfaces. A device interface serves as a symbolic link to your device stack's PDO. Once way to control access to the PDO is by specifying an SDDL string in your INF. If the SDDL string is not in the INF file, Windows will apply a default security descriptor. For more information, see Securing Device Objects and SDDL for Device Objects.

This section contains the following subsections:

NT Device Names

6/25/2019 • 2 minutes to read • Edit Online

A named device object has a name of the form **\Device***DeviceName*. This is known as the *NT device name* of the device object.

Device Names for WDM Drivers

WDM drivers do not name their device objects directly. Instead, the system imposes a uniform naming scheme that ensures that device names do not conflict between drivers. The naming scheme for WDM drivers is as follows.

- The PDO for a device is named. The bus driver requests named PDOs for the devices it enumerates. The bus driver specifies the FILE_AUTOGENERATED_DEVICE_NAME device characteristic when it creates the device object. For more information, see Specifying Device Characteristics. The system then automatically generates the device name.
- FDOs and filter DOs are not named. Function and filter drivers do not request a name when creating the device object.

Any I/O request to a named device object automatically goes to the top object in that device object's stack. Thus, only the PDO is required to be named. User-mode applications do not refer to WDM device objects by name; instead, applications access the device object through its *device interface*. For more information, see Device Interface Classes.

Driver writers must not name more than one object in a device stack. The operating system checks security settings based on the named object. If two different objects are named and have different security descriptors, the I/O requests that are sent to the object with the weaker security descriptor can reach the device object with the stronger security descriptor.

Device Names for non-WDM Drivers

A non-WDM driver must explicitly specify a name for any named device objects. The driver must create at least one named device object in the **\Device** object directory to receive I/O requests. The driver specifies the device name as the *DeviceName* parameter to **IoCreateDeviceSecure** when creating the device object.

Introduction to MS-DOS Device Names

6/25/2019 • 2 minutes to read • Edit Online

A named device object that is created by a non-WDM driver typically has an MS-DOS device name. An MS-DOS device name is a symbolic link in the object manager with a name of the form **\DosDevices***DosDeviceName*.

An example of a device with an MS-DOS device name is the serial port, COM1. It has the MS-DOS device name **\DosDevices\COM1**. Likewise, the C drive has the name **\DosDevices\C:**.

WDM drivers do not usually supply MS-DOS device names for their devices. Instead, WDM drivers use the **IoRegisterDeviceInterface** routine to register a device interface. The device interface specifies devices by their capabilities, rather than by a particular naming convention. For more information, see Device Interface Classes.

Drivers are required to supply an MS-DOS device name only if the device is required to have a specific well-known MS-DOS device name to work with user-mode programs.

A driver supplies an MS-DOS device name for a device object by using the **IoCreateSymbolicLink** routine to create a symbolic link to the device. For example, the following code example creates a symbolic link from **\DosDevices***DosDeviceName* to **\Device***DeviceName*.

```
UNICODE_STRING DeviceName;
UNICODE_STRING DosDeviceName;
NTSTATUS status;

RtlInitUnicodeString(&DeviceName, L"\\Device\\DeviceName");
RtlInitUnicodeString(&DosDeviceName, L"\\DosDevices\\DosDeviceName");
status = IoCreateSymbolicLink(&DosDeviceName, &DeviceName);
if (!NT_SUCCESS(status)) {
  /* Symbolic link creation failed.  Handle error appropriately. */
}
```

Note that the system supports multiple versions of the **\DosDevices** directory. Make sure that your driver creates its symbolic links in the version that you intend. For more information, see Local and Global MS-DOS Device Names.

To access the **DosDevices** namespace from user mode, specify **\\.** when you open a file name. You can open a corresponding device in user mode by calling **CreateFile()**.

For example, the following code example opens the \\DosDevices*DosDeviceName* device in user mode.

```
file = CreateFileW(L"\\\\.\\DosDeviceName",
  GENERIC READ | GENERIC WRITE,
    0,
    NULL,
    OPEN_EXISTING,
    0,
    NULL);
```

A symbolic link can also be created from a user-mode application by using the user-mode **DefineDosDevice** routine. For more information, see the Microsoft Windows SDK.

Local and Global MS-DOS Device Names

12/5/2018 • 3 minutes to read • Edit Online

The Microsoft Windows 2000 and later versions of the Windows NT-based operating system maintain multiple versions of the **DosDevices** directory.

On these operating systems, there is one *global* **\DosDevices** directory and multiple *local* **\DosDevices** directories. The global **\DosDevices** directory holds the MS-DOS device names that are visible system-wide. A local **\DosDevices** directory holds MS-DOS device names that are visible only in a particular *local* **DosDevices** *context*.

The local **DosDevices** contexts are as follows.

- On Windows XP and later, each logon session has its own local **DosDevices** context. System threads, and any thread that is running as the LocalSystem user, do not run in a local **DosDevices** context.
- On Windows 2000, each terminal server session has its own local **DosDevices** context. Any thread that is running as part of the console session does not run in a local **DosDevices** context.

Each thread has a current **DosDevices** context, which can change over the lifetime of a thread. A thread that does not run in a local **DosDevices** context is said to run in the *global* **DosDevices** *context*. Thus, the system account runs in the global **DosDevices** context.

If a thread is currently running in a local **DosDevices** context, any MS-DOS device names that it creates are created only in the local **DosDevices** directory. Thus, threads that are running in a local **DosDevices** context cannot affect the MS-DOS device names that are visible to threads that are running in another local **DosDevices** context or in the global **DosDevices** context. For example, if a user on Windows XP or later mounts a network drive as **X:**, this does not affect the meaning of **X:** for any other user, or for the system as a whole.

On Windows XP and later, when the object manager looks up a name in **\DosDevices**, it first searches the local **\DosDevices** directory, and then the global **\DosDevices** directory. If the name exists in both places, the local name shadows the global name.

On Windows 2000, whenever a new terminal server session is initiated, the system builds local **DosDevices** directory by copying the global **\DosDevices** directory. Any subsequent changes to the global directory are not propagated to the local directory.

A driver that must create its MS-DOS device names in the global **\DosDevices** directory can do so by creating its symbolic links in a standard driver routine that is guaranteed to run in a system thread context, such as **DriverEntry**. Alternatively, the global **\DosDevices** directory is available as **\DosDevices\Global**; drivers can use a name of the **\DosDevices\Global***DosDeviceName* to specify a name in the global directory.

Note that **\DosDevices\Global** does not exist on platforms that do not support local and global versions of **\DosDevices**, such as Windows 98/Me. The following code example creates a global symbolic link that works on Windows 98/Me as well as Windows 2000 and later operating systems:

```
UNICODE_STRING deviceName; // Already initialized.
UNICODE_STRING symbolicLinkName; // Initializing below.
NTSTATUS status;

if (IoIsWdmVersionAvailable(1, 0x10)) {
    // We're on Windows 2000 or later, so we use \DosDevices\Global.

    RtlInitUnicodeString(&symbolicLinkName, L"\\DosDevices\\Global\\SymbolicLinkName");

} else {
    // Windows 98/Me.  We just use DosDevices.

    RtlInitUnicodeString(&symbolicLinkName, L"\\DosDevices\\SymbolicLinkName");
}

status = IoCreateSymbolicLink(&symbolicLinkName, &deviceName);
if (!NT_SUCCESS(status)) {
  /* Symbolic link creation failed.  Handle error appropriately. */
}
```

A driver can create MS-DOS device names in a local **\DosDevices** directories by creating the symbolic link in response to an IOCTL. When a thread in a particular local **DosDevices** context sends the IOCTL, the driver's *DispatchDeviceControl* is called from within the current thread context.

For more information about the context in which a standard driver routine runs, see Dispatch Routines and IRQLs.

The system distinguishes local **\DosDevices** directories as follows:

- On Windows XP and later, local **\DosDevices** directories are identified by the **AuthenticationID** for the logon session's access token. For more information about the **AuthenticationID**, see the description of the **TOKEN_STATISTICS** structure in the Microsoft Windows SDK documentation.
- On Windows 2000, local **\DosDevices** directories are identified by the **SessionId** for the terminal server session. For more information about the **SessionId**, see the description of the **WTS_SESSION_INFO** structure in the Windows SDK documentation.

Windows NT 4.0 Terminal Server Edition supports local **DosDevices** directories in the exact same manner as Windows 2000.

Device Extensions

6/25/2019 • 3 minutes to read • Edit Online

For most intermediate and lowest-level drivers, the device extension is the most important data structure associated with a device object. Its internal structure is driver-defined, and it is typically used to:

- Maintain device state information.
- Provide storage for any kernel-defined objects or other system resources, such as spin locks, used by the driver.
- Hold any data the driver must have resident and in system space to carry out its I/O operations.

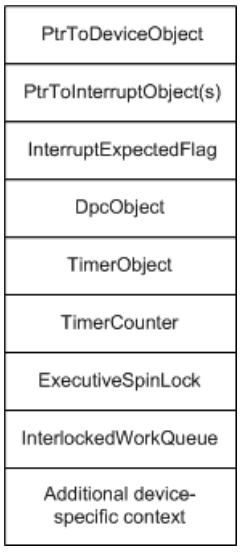
Because most bus, function, and filter drivers (lowest-level and intermediate drivers) execute in an arbitrary thread context (that of whatever thread happens to be current), a device extension is each driver's primary place to maintain device state and all other device-specific data the driver needs. For example, any driver that implements a *CustomTimerDpc* or *CustomDpc* routine usually provides storage for the required kernel-defined timer and/or DPC objects in a device extension.

Every driver that has an ISR must provide storage for a pointer to a set of kernel-defined interrupt objects, and most device drivers store this pointer in a device extension. Each driver determines the size of the device extension when it creates a device object, and each driver defines the contents and structure of its own device extensions.

The I/O manager's **IoCreateDevice** and **IoCreateDeviceSecure** routines allocate memory for the device object and extension from the nonpaged memory pool.

Every standard driver routine that receives an IRP also receives a pointer to a device object representing the target device for the requested I/O operation. These driver routines can access the corresponding device extension through this pointer. Usually, a *DeviceObject* pointer is also an input parameter to a lowest-level driver's ISR.

The following figure shows a representative set of driver-defined data for the device extension of a lowest-level driver's device object. A higher-level driver would not provide storage for an interrupt object pointer returned by **IoConnectInterrupt** and passed to **KeSynchronizeExecution** and **IoDisconnectInterrupt**. However, a higher-level driver would provide storage for the timer and DPC objects shown in the following figure if the driver has a *CustomTimerDpc* routine. A higher-level driver also might provide storage for an executive spin lock and interlocked work queue.



In addition to providing storage for an interrupt object pointer, a lowest-level device driver must supply storage for an interrupt spin lock if its ISR handles interrupts for two or more devices on different vectors or if it has more than one ISR. For more information about registering an ISR, see Registering an ISR.

Typically, drivers store pointers to their device objects in their device extensions, as shown in the figure. A driver might also keep a copy of the resource list for the device in the extension.

A higher-level driver typically stores a pointer to the next-lower driver's device object in its device extension. A higher-level driver must pass a pointer to the next-lower driver's device object to **IoCallDriver**, after it has set up the next-lower driver's I/O stack location in an IRP, as explained in Handling IRPs.

Note also that any higher-level driver that allocates IRPs for lower-level drivers must specify how many stack locations the new IRPs should have. In particular, if a higher-level driver calls **IoMakeAssociatedIrp**, **IoAllocateIrp**, or **IoInitializeIrp**, it must access the target device object of the next-lower-level driver to read its **StackSize** value, in order to supply the correct *StackSize* as an argument to these support routines.

While a higher-level driver can read data from the next-lower-level driver's device object through the pointer returned by **IoAttachDeviceToDeviceStack**, such a driver must follow these implementation guidelines:

- Never attempt to write data to the lower driver's device object.

  The only exceptions to this guideline are file systems, which set and clear DO_VERIFY_VOLUME in the **Flags** of lower-level removable-media drivers' device objects.

- Never attempt to access the lower driver's device extension for the following reasons:

  - There is no safe way to synchronize access to a single device extension between two drivers.
  - A pair of drivers that implement such a backdoor communication scheme cannot be upgraded individually, cannot have an intermediate driver inserted between them without changing existing driver source, and cannot be recompiled and moved readily from one Windows platform to the next.

To preserve their interoperability with lower-level drivers from one Windows platform or version to the next, higher-level drivers either must reuse the IRPs given them or must create new IRPs, and they must use **IoCallDriver** to communicate requests to lower-level drivers.

Properties of Device Objects

12/5/2018 • 2 minutes to read • Edit Online

Each device object has certain properties that describe the device and how the device object interacts with the system. The device object properties include:

- Device type. Specifies the device's type of hardware. For more information about device types, see Specifying Device Types.
- Device characteristics. Specifies flags that provide additional information about the device. For more information, see Specifying Device Characteristics.
- Exclusive access. Specifies whether the device object represents an *exclusive device*. If the device is exclusive, only one handle can be open for the device object at a time. (If the underlying device supports overlapped I/O, multiple threads of the same process can send requests through a single handle.) For more information, see Specifying Exclusive Access to Device Objects.
- Security descriptor. Device objects have a security descriptor that controls access to the device. For more information, see Securing Device Objects.

For each of these properties, a default value can be set when the device object is created. For more information about creating device objects, see Creating a Device Object.

Values for device object properties can also be set in the registry. See Setting Device Object Properties in the Registry for more information.

Specifying Device Types

6/25/2019 • 2 minutes to read • Edit Online

Each device object has a *device type*, which is stored in the **DeviceType** member of its **DEVICE_OBJECT** structure. The device type represents the type of underlying hardware for the driver.

Every kernel-mode driver that creates a device object must specify an appropriate device type value when calling **IoCreateDevice**. The **IoCreateDevice** routine uses the supplied device type to initialize the **DeviceType** member of the **DEVICE_OBJECT** structure.

The system defines the following device type values, listed in alphabetical order:

```
#define FILE_DEVICE_8042_PORT           0x00000027
#define FILE_DEVICE_ACPI                0x00000032
#define FILE_DEVICE_BATTERY             0x00000029
#define FILE_DEVICE_BEEP                0x00000001
#define FILE_DEVICE_BUS_EXTENDER        0x0000002a
#define FILE_DEVICE_CD_ROM              0x00000002
#define FILE_DEVICE_CD_ROM_FILE_SYSTEM  0x00000003
#define FILE_DEVICE_CHANGER             0x00000030
#define FILE_DEVICE_CONTROLLER          0x00000004
#define FILE_DEVICE_DATALINK            0x00000005
#define FILE_DEVICE_DFS                 0x00000006
#define FILE_DEVICE_DFS_FILE_SYSTEM     0x00000035
#define FILE_DEVICE_DFS_VOLUME          0x00000036
#define FILE_DEVICE_DISK                0x00000007
#define FILE_DEVICE_DISK_FILE_SYSTEM    0x00000008
#define FILE_DEVICE_DVD                 0x00000033
#define FILE_DEVICE_FILE_SYSTEM         0x00000009
#define FILE_DEVICE_FIPS                0x0000003a
#define FILE_DEVICE_FULLSCREEN_VIDEO    0x00000034
#define FILE_DEVICE_INPORT_PORT         0x0000000a
#define FILE_DEVICE_KEYBOARD            0x0000000b
#define FILE_DEVICE_KS                  0x0000002f
#define FILE_DEVICE_KSEC                0x00000039
#define FILE_DEVICE_MAILSLOT            0x0000000c
#define FILE_DEVICE_MASS_STORAGE        0x0000002d
#define FILE_DEVICE_MIDI_IN             0x0000000d
#define FILE_DEVICE_MIDI_OUT            0x0000000e
#define FILE_DEVICE_MODEM               0x0000002b
#define FILE_DEVICE_MOUSE               0x0000000f
#define FILE_DEVICE_MULTI_UNC_PROVIDER  0x00000010
#define FILE_DEVICE_NAMED_PIPE          0x00000011
#define FILE_DEVICE_NETWORK             0x00000012
#define FILE_DEVICE_NETWORK_BROWSER     0x00000013
#define FILE_DEVICE_NETWORK_FILE_SYSTEM 0x00000014
#define FILE_DEVICE_NETWORK_REDIRECTOR  0x00000028
#define FILE_DEVICE_NULL                0x00000015
#define FILE_DEVICE_PARALLEL_PORT       0x00000016
#define FILE_DEVICE_PHYSICAL_NETCARD    0x00000017
#define FILE_DEVICE_PRINTER             0x00000018
#define FILE_DEVICE_SCANNER             0x00000019
#define FILE_DEVICE_SCREEN              0x0000001c
#define FILE_DEVICE_SERENUM             0x00000037
#define FILE_DEVICE_SERIAL_MOUSE_PORT   0x0000001a
#define FILE_DEVICE_SERIAL_PORT         0x0000001b
#define FILE_DEVICE_SMARTCARD           0x00000031
#define FILE_DEVICE_SMB                 0x0000002e
#define FILE_DEVICE_SOUND               0x0000001d
#define FILE_DEVICE_STREAMS             0x0000001e
#define FILE_DEVICE_TAPE                0x0000001f
#define FILE_DEVICE_TAPE_FILE_SYSTEM    0x00000020
#define FILE_DEVICE_TERMSRV             0x00000038
#define FILE_DEVICE_TRANSPORT           0x00000021
#define FILE_DEVICE_UNKNOWN             0x00000022
#define FILE_DEVICE_VDM                 0x0000002c
#define FILE_DEVICE_VIDEO               0x00000023
#define FILE_DEVICE_VIRTUAL_DISK        0x00000024
#define FILE_DEVICE_WAVE_IN             0x00000025
#define FILE_DEVICE_WAVE_OUT            0x00000026
```

These constants are defined in Ntddk.h and Wdm.h. Check these files to see whether additional device types have been defined.

The FILE_DEVICE_DISK specification covers disk partitions and any object that appears as a disk.

Intermediate drivers usually specify device types that represent the underlying device. For example, the system-supplied fault-tolerant disk driver, *ftdisk*, creates device objects of type FILE_DEVICE_DISK; it does not define new

device types for the mirror sets, stripe sets, and volume sets it manages.

FILE_DEVICE_*XXX* values in the range of 0 through 32767 are reserved for Microsoft. All driver writers must use these system-defined constants for devices belonging to the system-defined device types.

If a type of hardware does not match any of the defined types, specify a value of either FILE_DEVICE_UNKNOWN, or a value within the range of 32768 through 65535.

Specifying Device Characteristics

6/25/2019 • 2 minutes to read • Edit Online

Each device object can have one or more device characteristics. Device characteristics are stored as flags in the **Characteristics** member of the device object's **DEVICE_OBJECT** structure.

Most drivers specify only the FILE_DEVICE_SECURE_OPEN characteristic. This ensures that the same security settings are applied to any open request into the device's namespace. For more information, see Controlling Device Namespace Access.

The FILE_AUTOGENERATED_DEVICE_NAME is only used for PDOs. The FILE_FLOPPY_DISKETTE, FILE_REMOVABLE_MEDIA, and FILE_WRITE_ONCE_MEDIA characteristics are specific to storage devices. For a description of the possible device characteristic flags, see the description of the **Characteristics** member of **DEVICE_OBJECT**.

Certain device characteristics, such as FILE_AUTOGENERATED_DEVICE_NAME, only apply to individual device objects. Drivers can specify a setting for the device characteristics for individual device objects when they create the device object by calling **IoCreateDevice** or **IoCreateDeviceSecure**.

The following characteristics apply to the entire device stack:

FILE_DEVICE_SECURE_OPEN

FILE_FLOPPY_DISKETTE

FILE_READ_ONLY_DEVICE

FILE_REMOVABLE_MEDIA

FILE_WRITE_ONCE_MEDIA

Drivers can set device characteristics that apply to the entire device stack by calling **IoCreateDevice** or **IoCreateDeviceSecure**. Alternatively, device characteristics that apply to the entire device stack can be set in the registry, for either the device or for the device's setup class. (For more information, see Setting Device Object Properties in the Registry.)

The PnP manager determines the registry setting for device characteristics as follows.

- If a value is specified for the individual device, the PnP manager uses that value;
- Otherwise, if a value is specified for the device setup class, the PnP manager uses that value;
- Otherwise, the PnP manager uses a value of zero as the registry setting.

If a device characteristic that applies to the entire device stack is set in the registry, or if it is set for any FDO or filter DO in the stack, then the PnP manager sets it for every device object in the stack. (If the device is *raw mode* capable, and thus does not have an FDO, then the PnP manager uses the PDO instead.)

Specifying Exclusive Access to Device Objects

6/25/2019 • 2 minutes to read • Edit Online

If exclusive access to a device is enabled, only one handle to the device can be open at a time. For the I/O manager to enforce exclusive access to the device, the exclusive property must be set for the named device object in the device stack.

For a WDM device stack that has a both a PDO and an FDO, the exclusive property can be set only by the INF file, by using an **INF AddReg directive**. The PDO is the named object in the stack, but the bus driver (not the function driver itself) creates the PDO, on behalf of the function driver. The only way to direct the bus driver to set the exclusive flag for the PDO is by the class or device INF files. (The call to the **IoCreateDevice** routine creates the FDO; setting the exclusive flag for the FDO has no effect.)

Drivers whose device objects are not stacked, such as non-WDM drivers and devices that operate in raw mode, can use the **IoCreateDeviceSecure** routine to set the exclusive property for their named device object.

The I/O manager enforces exclusivity on a per name basis on named device objects, regardless of the trailing name. For example, suppose the device object has the name "\Device\DeviceName". Then, the I/O manager enforces exclusivity for a request to open "\Device\DeviceName*Filename1*" followed by "\Device\DeviceName*Filename2*". If two objects in the device stack are named (which is not recommended), the I/O manager allows a single handle to be opened for each object. In such a situation, drivers must enforce exclusivity themselves within their *DRIVER_DISPATCH* callback functions. The I/O manager also does not enforce exclusivity for opens relative to another file handle. For more information about file open requests in the device's namespace, see Controlling Device Namespace Access.

Setting Device Object Properties in the Registry

6/25/2019 • 2 minutes to read • Edit Online

Properties of device objects can be set in the registry as follows:

- For WDM drivers, properties can be set for each model of a device, or for a whole device setup class. (For more information about device setup classes, see Device Setup Classes.)
- For non-WDM drivers, properties can be set for a named device object's device setup class. The driver specifies the device setup class when it creates the device object with **IoCreateDeviceSecure**. For more information about how to specify a device setup class, see **IoCreateDeviceSecure**.

Any settings in the registry override the properties supplied when the driver created the device object.

Registry settings are specified by an INF file that is used during device installation, or they can be specified after installation by an application that calls the device installation functions.

This section contains the following subsections:

Setting Device Object Registry Properties During Installation

6/25/2019 • 2 minutes to read • Edit Online

To set device object properties during installation, you must provide an INF file that specifies the properties. You can specify device object properties for either a device, or a device setup class.

These are specified as follows.

- For an individual device, properties are set in the *add-registry-section* for the device. The INF **AddReg** directive within the device's *DDInstall*.HW section specifies the *add-registry-section* for the device.
- For a device setup class, properties are set in the *add-registry-section* for the device setup class. The INF **AddReg** directive within the **ClassInstall32** section for the class specifies the *add-registry-section* for the class.

Within an *add-registry-section*, the following keywords can be used to specify the individual device object property to set.

KEYWORD	DEVICE OBJECT PROPERTY
DeviceType	Device type
DeviceCharacteristics	Device characteristics
Exclusive	Exclusive
Security	Security descriptor

For more information about using these keywords, see **INF AddReg Directive**.

The settings can be set by a user-mode component by using the device installation functions. For more information, see Setting Device Object Registry Properties After Installation.

Setting Device Object Registry Properties After Installation

6/25/2019 • 2 minutes to read • Edit Online

A user-mode program can use the device installation functions to get or set the registry settings for the properties of a driver's device object. Normally these functions are used by installation software, but they can be used by any user-mode program. (The program must be executed by a user that has Administrator access.)

The **SetupDiGetDeviceRegistryProperty** and **SetupDiSetDeviceRegistryProperty** functions get and set the registry key for each specified property. The *Property* parameter specifies the property to get or set. The *PropertyBuffer* points to the destination buffer (when getting the property) or source buffer (when setting the property) for the property.

The correspondence between values for the *Property* parameter and actual properties is as follows.

VALUE FOR *PROPERTY* PARAMETER	DEVICE OBJECT PROPERTY
SPDRP_CHARACTERISTICS	Device characteristics
SPDRP_DEVTYPE	Device type
SPDRP_EXCLUSIVE	Exclusive
SPDRP_SECURITY	Security descriptor as a **SECURITY_DESCRIPTOR** structure
SPDRP_SECURITY_SDS	Security descriptor as an SDDL string

Note that two different ways are provided to get or set the security descriptor. You can specify the SPDRP_SECURITY value to treat the security descriptor as a **SECURITY_DESCRIPTOR** structure, or SPDRP_SECURITY_SDS to treat the security descriptor as an SDDL string. For more information about SDDL strings, see SDDL for Device Objects.

For Windows XP and later operating systems, programs can also get and set the property values for a device setup class. Use the **SetupDiGetClassRegistryProperty** and **SetupDiSetClassRegistryProperty** functions to get and set the property values for a device setup class.

For more information about using the **SetupDi*Xxx*** functions, see Using Device Installation Functions.

Points to Consider About Device Objects

6/25/2019 • 2 minutes to read • Edit Online

Keep the following points in mind when designing a kernel-mode driver:

- Except for certain file system drivers, all I/O operations are always sent to the top device object of a device stack.
- Device stacks are identified using the name of the named device object in the stack, or by using an alias for that name, such as a symbolic link or a device interface. For WDM function drivers, the named device object is created by the bus driver for the device. Non-WDM drivers must create their own named device objects.
- A lowest-level driver, such as a PnP hardware bus driver, creates a physical device object (PDO) for each device it controls. An intermediate driver, such as a PnP function driver, creates a functional device object (FDO).

  A WDM driver creates device objects in its *AddDevice* routine, which is called by the PnP manager after device enumeration.
- For most lowest-level and intermediate drivers, the device extension of each device object is each driver's primary (and frequently only) global data storage area. Many drivers maintain device state and all other device-specific data and resources a driver requires in the driver-defined device extension of each driver-created device object.

  (Additionally, the driver-specific I/O stack location associated with an IRP can be considered an operation-specific local storage area for some kinds of data.)

For more information about the device objects a specific driver must create, see the device-type-specific documentation in the Windows Driver Kit (WDK).

Managing Kernel Objects

12/5/2018 • 2 minutes to read • Edit Online

The Windows Object Manager controls *objects* that are part of the kernel-mode operating system. An object is a collection of data that the operating system manages.

Typical kernel-mode objects include the following objects:

- Device objects (See Device Objects and Device Stacks.)
- File objects.
- Symbolic links.
- Registry keys.
- Threads and processes.
- Kernel dispatcher objects, such as event objects and mutex objects. (See Kernel Dispatcher Objects.)
- Callback objects. (See Callback Objects.)
- Section objects. (See Section Objects and Views.)

Kernel-mode objects enable you to manipulate objects in partnership with the object manager without damaging the portions of the objects that the operating system needs. This principle is called *encapsulation* and is one of the core concepts of object-oriented programming. (Because kernel-mode objects do not provide other aspects of object-orientation, kernel-mode programming is typically referred to as *object-based*.) Kernel-mode objects do not follow the same rules as objects in C++ or Microsoft COM.

Kernel-mode objects can be referenced by pointers. An object may have an object name. For more information about object names, see Object Names.

User-mode programmers can reference objects only through indirection, using a *handle*. If an object has a name, you can use it to obtain the handle in user mode. For more information about handles, see Object Handles.

Kernel-mode objects have a very specific life-cycle. For more information about object life-cycles, see Life Cycle of an Object.

Object security is a prime concern for kernel-mode programming. For more information on object security, see Object Security.

The kernel-mode environment stores objects in a virtual directory system, also known as the object namespace. This allows objects to be accessed in a hierarchical way with parent and child objects. This namespace is similar to a file system set of directories but does not exactly correspond to a particular file system on your computer. For more information about object directories, see Object Directories.

Object Names

7/18/2019 • 2 minutes to read • Edit Online

Kernel-mode objects are either named or unnamed. The *object name* is a Unicode string that both user-mode and kernel-mode components can use to refer to the object. For example, **\KernelObjects\LowMemoryCondition** is the name of the standard event object that signals when the amount of free memory in the system is low.

Both user-mode and kernel-mode components use the object name to open a handle to an object. All subsequent operations are performed by using the handle.

If an object is unnamed, a user-mode component cannot open a handle to it. Kernel-mode components can refer to an unnamed object by either a pointer or a handle.

Named objects are organized into a hierarchy. Each object is named relative to a parent object. Each component of the object's name begins with a backslash character. For example, **\KernelObjects** is the parent object for **\KernelObjects\LowMemoryCondition**.

Only some types of objects can have child objects. The following are some examples:

- Object directories have child objects. The object manager uses object directories to organize objects. For example **\KernelObjects** is an object directory that holds standard event objects. Object directories do not correspond to actual directories on a disk. For more information, see Object Directories.
- Device objects for disk drives have child objects that correspond to files on the disk.
- File objects that represent directories have child objects corresponding to files within the directory.
- Device objects for WDM drivers have their own namespace that can be used in a driver-defined fashion. For more information, see Controlling Device Namespace Access.

Files have object names that are relative to **\DosDevices**. For example, the file C:\Directory\File can be specified as **\DosDevices\C:***Directory**File*.

For example, the components of the object name can be described as follows.

OBJECT NAME	DESCRIPTION
\DosDevices	Object directory.
\DosDevices\C:	Device object representing the C: drive.
\DosDevices\C:*Directory*	File object representing the directory named C:\Directory.
\DosDevices\C:*Directory**File*	File object representing the file named C:\Directory\File.

Drivers that create named objects do so in specific object directories. For more information, see Object Directories.

Object Directories

6/25/2019 • 2 minutes to read • Edit Online

An *object directory* is a named object that is used solely to contain other named objects. For example, the **\Device** object directory contains the named device objects created by drivers.

Do not confuse object directories with file system directories. Object directories exist only within the object manager, and do not correspond to any directory on disk. (File system directories are, in fact, represented as file objects.)

The following is a list of the top-level object directories that contain objects drivers might create or use:

- **\Callbacks**

  The system creates standard callback objects in this directory. For more information, see Using a System-Defined Callback Object.

- **\Device**

  Drivers create named device objects in this directory. For more information, see Named Device Objects.

- **\KernelObjects**

  The system creates standard event objects in this directory. For more information, see Standard Event Objects.

- **\DosDevices**

  This directory stores the MS-DOS device name of a device as a symbolic link to the corresponding device object. For more information, see MS-DOS Device Names.

The system creates other top-level directories, but they are reserved for system use.

Drivers can create new object directories by calling the **ZwCreateDirectoryObject** routine.

Life Cycle of an Object

6/25/2019 • 2 minutes to read • Edit Online

This topic describes the "life cycle" of an object, that is, how objects are referenced and tracked by the object manager. This topic also describes how to make objects temporary or permanent.

Object Reference Count

The object manager maintains a count of the number of references to an object. When an object is created, the object manager sets the object's reference count to one. Once that counter falls to zero, the object is freed.

Drivers must ensure that the object manager has an accurate reference count for any objects they manipulate. An object that is released prematurely can cause the system to crash. An object whose reference count is mistakenly high will never be freed.

Objects can be referenced either by handle, or by pointer. In addition to the reference count, the object manager maintains a count of the number of open handles to an object. Each routine that opens a handle increases both the object reference count and the object handle count by one. Each call to such a routine must be matched with a corresponding call to **ZwClose**. For more information, see Object Handles.

Within kernel mode, objects can be referenced by a pointer to the object. Routines that return pointers to objects, such as **IoGetAttachedDeviceReference**, increase the reference count by one. Once the driver is done using the pointer, it must call **ObDereferenceObject** to decrease the reference count by one.

The following routines all increase the reference count of an object by one:

ExCreateCallback

IoGetAttachedDeviceReference

IoGetDeviceObjectPointer

IoWMIOpenBlock

ObReferenceObject

ObReferenceObjectByHandle

ObReferenceObjectByPointer

Each call that is made to any of the preceding routines must be matched with a corresponding call to **ObDereferenceObject**.

The **ObReferenceObject** and **ObReferenceObjectByPointer** routines are provided so that drivers can increase the reference count of a known object pointer by one. **ObReferenceObject** simply increases the reference count. **ObReferenceObjectByPointer** does an access check before increasing the reference count.

The **ObReferenceObjectByHandle** routine receives an object handle and supplies a pointer to the underlying object. It too increases the reference count by one.

Temporary and Permanent Objects

Most objects are *temporary*; they exist as long as they are in use, and then they are freed by the object manager. Objects can be created that are *permanent*. If an object is permanent, the object manager itself holds a reference to the object. Thus, its reference count remains greater than zero, and the object is not freed when it is no longer in use.

A temporary object can be accessed by name only as long as its handle count is nonzero. Once the handle count decrements to zero, the object's name is removed from the object manager's namespace. Such objects can still be accessed by pointer as long as their reference count remains greater than zero. Permanent objects can be accessed by name as long as they exist.

An object can be made permanent at the time of its creation by specifying the OBJ_PERMANENT attribute in the **OBJECT_ATTRIBUTES** structure for the object. For more information, see **InitializeObjectAttributes**.

To make a permanent object temporary, use the **ZwMakeTemporaryObject** routine. This routine causes an object to be automatically deleted once it is no longer in use. (If the object has no open handles, the object's name is immediately removed from the object manager's namespace. The object itself remains until the reference count falls to zero.)

Object Handles

6/25/2019 • 2 minutes to read • Edit Online

Drivers and user-mode components access most system-defined objects through *handles*. Handles are represented by the HANDLE opaque data type. (Note that handles are not used to access device objects or driver objects.)

For most object types, the kernel-mode routine that creates or opens the object provides a handle to the caller. The caller then uses that handle in subsequent operations on the object.

Here is a list of object types that drivers typically use, and the routines that provide handles to objects of that type.

OBJECT TYPE	CORRESPONDING CREATE/OPEN ROUTINE
File	**IoCreateFile**, **ZwCreateFile**, **ZwOpenFile**
Registry keys	**IoOpenDeviceInterfaceRegistryKey**, **IoOpenDeviceRegistryKey**, **ZwCreateKey**, **ZwOpenKey**
Threads	**PsCreateSystemThread**
Events	**IoCreateSynchronizationEvent**, **IoCreateNotificationEvent**
Symbolic links	**ZwOpenSymbolicLinkObject**
Directory objects	**ZwCreateDirectoryObject**
Section objects	**ZwOpenSection**

When the driver no longer requires access to the object, it calls the **ZwClose** routine to close the handle. This works for all of the object types listed in the table above.

Most of the routines that provide handles take an **OBJECT_ATTRIBUTES** structure as a parameter. This structure can be used to specify attributes for the handle.

Drivers can specify the following handle attributes:

- OBJ_KERNEL_HANDLE

  The handle can only be accessed from kernel mode.

- OBJ_INHERIT

  Any children of the current process receive a copy of the handle when they are created.

- OBJ_FORCE_ACCESS_CHECK

This attribute specifies that the system performs all access checks on the handle. By default, the system bypasses all access checks on handles created in kernel mode.

Use the **InitializeObjectAttributes** routine to set these attributes in an **OBJECT_ATTRIBUTES** structure.

For information about validating object handles, see Failure to Validate Object Handles.

Private Object Handles

Whenever a driver creates an object handle for its private use, the driver must specify the OBJ_KERNEL_HANDLE attribute. This ensures that the handle is inaccessible to user-mode applications.

Shared Object Handles

A driver that shares object handles between kernel mode and user mode must be carefully written to avoid accidentally creating security holes. Here are some guidelines:

1. Create handles in kernel mode and pass them to user mode, instead of the other way around. Handles created by a user-mode component and passed to the driver should not be trusted.
2. If the driver must manipulate handles on behalf of user-mode applications, use the OBJ_FORCE_ACCESS_CHECK attribute to verify that the application has the necessary access.
3. Use **ObReferenceObjectByPointer** to keep a kernel-mode reference on a shared handle. Otherwise, if a user-mode component closes the handle, the reference count goes to zero, and if the driver then tries to use or close the handle the system will crash.

If a user-mode application creates an event object, a driver can safely wait for that event to be signaled, but only if the application passes a handle to the event object to the driver through an IOCTL. The driver must handle the IOCTL in the context of the process that created the event and must validate that the handle is an event handle by calling **ObReferenceObjectByHandle**.

Memory Management for Windows Drivers

10/17/2019 • 2 minutes to read • Edit Online

Kernel-mode drivers allocate memory for purposes such as storing internal data, buffering data during I/O operations, and sharing memory with other kernel-mode and user-mode components. Driver developers should understand memory management in Windows so that they use allocated memory correctly and efficiently. Windows manages virtual and physical memory, and divides memory into separate user and system address spaces. A driver can specify whether allocated memory supports capabilities such as demand paging, data caching, and instruction execution.

The *memory manager* is the kernel component that performs the memory management operations in Windows. For more information, see Windows Kernel-Mode Memory Manager.

The memory manager implements a number of kernel-mode support routines that drivers call to allocate and manage memory. For more information, see Memory Allocation and Buffer Management.

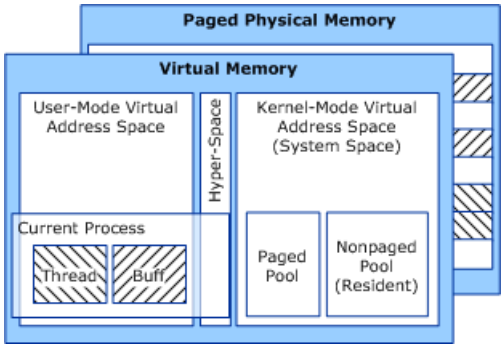
The memory-management capabilities of kernel-mode drivers are different from those of user-mode applications. For more information about memory management for applications, see Memory Management.

In this section

Overview of Windows Memory Space

12/5/2018 • 2 minutes to read • Edit Online

The following figure illustrates the NT-based operating system's virtual memory spaces and their relationship to system physical memory.



As this figure shows, virtual memory is backed by paged physical memory, and a virtual address range can be backed by discontiguous physical memory pages. User-space virtual memory and system-space memory allocated from paged pool are always *pageable*. Any user-space code or data can be paged out to secondary storage at any time, even while the process is executing.

Note that any noncurrent process's virtual addresses are not visible, so its memory space is inaccessible.

For an extensive discussion of memory management, see the *Inside Microsoft Windows Internals* book from Microsoft Press.

Allocating System-Space Memory

6/25/2019 • 2 minutes to read • Edit Online

Drivers can use system-allocated space within their device extensions as global storage areas for device-specific information. Drivers can use only the kernel stack to pass small amounts of data to their internal routines. Some drivers have to allocate additional, larger amounts of system-space memory, typically for I/O buffers.

To allocate I/O buffer space, the best memory allocation routines to use are **MmAllocateNonCachedMemory**, **MmAllocateContiguousMemorySpecifyCache**, **AllocateCommonBuffer** (if the driver's device uses bus-master DMA or a system DMA controller's auto-initialize mode), or **ExAllocatePoolWithTag**.

Nonpaged pool typically becomes fragmented as the system runs, so a driver's **DriverEntry** routine should call these routines to set up any long-term I/O buffers the driver needs. Each of these routines, except **ExAllocatePoolWithTag**, allocates memory that is aligned on a processor-specific boundary (determined by the processor's data-cache-line size) to provide best performance.

Drivers should allocate I/O buffers as economically as possible, because nonpaged pool memory is a limited system resource. Typically, a driver should avoid calling these support routines repeatedly to request allocations of less than PAGE_SIZE because each allocation that is less than PAGE_SIZE also comes with a pool header that is used to internally manage the allocation.

Tips for Allocating Driver Buffer Space Economically

To allocate I/O buffer memory economically, be aware of the following:

- Each call to **MmAllocateNonCachedMemory** or **MmAllocateContiguousMemorySpecifyCache** always returns a full multiple of the system's page size, of nonpaged system-space memory, whatever the size of the requested allocation. Therefore, requests for less than a page are rounded up to a full page and any remainder bytes on the page are wasted; they are inaccessible by the driver that called the function and are unusable by other kernel-mode code.
- Each call to **AllocateCommonBuffer** uses at least one adapter object map register, which maps at least one byte and at most one page. For more information about map registers and using common buffers, see Adapter Objects and DMA.

Allocating Memory with ExAllocatePoolWithTag

Drivers can also call **ExAllocatePoolWithTag**, specifying one of the following system-defined **POOL_TYPE** values for the *PoolType* parameter:

- *PoolType* = **NonPagedPool** for any objects or resources not stored in a device extension or controller extension that the driver might access while it is running at IRQL > APC_LEVEL.

  For this *PoolType* value, **ExAllocatePoolWithTag** allocates the amount of memory that is requested if the specified *NumberOfBytes* is less than or equal to PAGE_SIZE. Otherwise, any remainder bytes on the last-allocated page are wasted: inaccessible to the caller and unusable by other kernel-mode code.

  For example, on an x86, an allocation request of 5 kilobytes (KB) returns two 4-KB pages. The last 3 KB of the second page is unavailable to the caller or another caller. To avoid wasting nonpaged pool, the driver should allocate multiple pages efficiently. In this case, for example, the driver could make two allocations, one for PAGE_SIZE and the other for 1 KB, to allocate a total of 5 KB.

  Note Starting with Windows Vista, the system automatically adds the additional memory so two allocations are unnecessary.

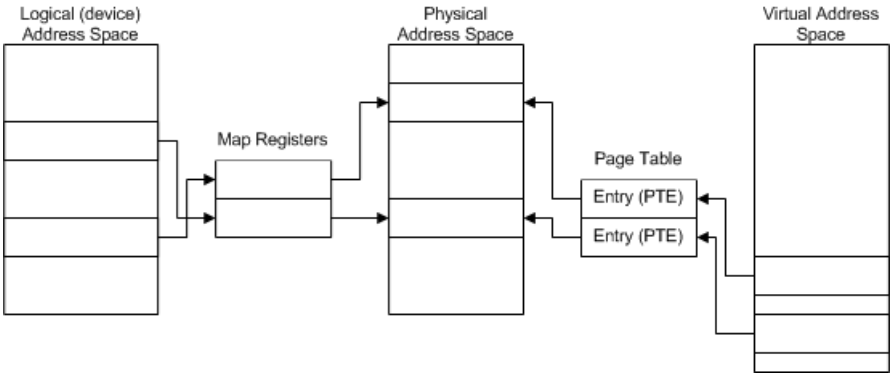
- *PoolType* = **PagedPool** for memory that is always accessed at IRQL <= APC_LEVEL and is not in the file system's write path.

ExAllocatePoolWithTag returns a **NULL** pointer if it cannot allocate the requested number of bytes. Drivers should always check the returned pointer. If its value is **NULL**, the **DriverEntry** routine (or any other driver routine that returns NTSTATUS values) should return STATUS_INSUFFICIENT_RESOURCES or handle the error condition if possible.

Map Registers

5/6/2019 • 4 minutes to read • Edit Online

Drivers that perform DMA use three different address spaces, as shown in the following figure.



On any Windows platform, a driver has access to the full virtual address space supported by the processor. On a 32-bit processor, the virtual address space represents four gigabytes. The CPU translates addresses in the virtual address space to addresses in the system's physical address space by using a page table. Each page table entry (PTE) maps one page of virtual memory to a page of physical memory, resulting in a paging operation when necessary. An MDL (memory descriptor list) provides a similar mapping for a buffer associated with driver DMA operations.

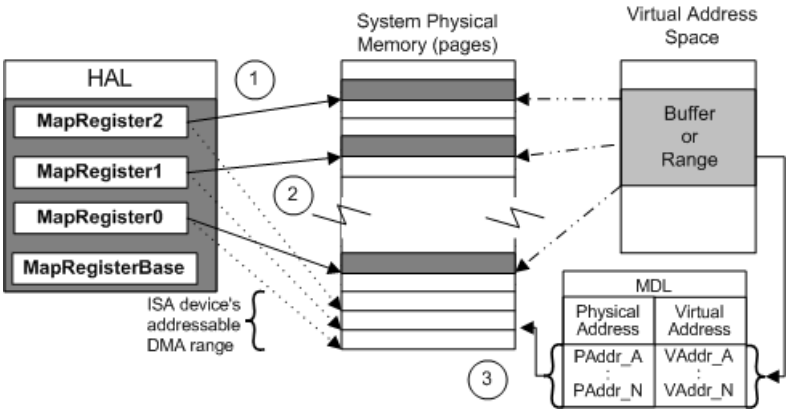
Devices vary in their ability to access the system's full virtual address space. A device uses addresses in logical (device) address space. Each HAL uses *map registers* to translate a device or logical address to a physical address (a location in physical RAM). For the device hardware, map registers perform the same function that the MDL (and page table) performs for the software (drivers): they translate addresses to physical memory.

Because these address spaces are separately addressed, a driver cannot use a pointer in virtual address space to address a location in physical memory, and vice versa. The driver must first translate the virtual address to a physical address. Similarly, a device cannot use a logical address to directly access physical memory. The device must first translate the address.

A HAL must set up adapter objects that support DMA for a wide variety of DMA devices and I/O buses on different computers. For example, most ISA DMA controllers, subordinate devices, and bus-master devices have insufficient address lines to access the full four-gigabyte system physical address space of a 32-bit processor (or the 64-gigabyte system physical address of an x86 processor running in 36-bit PAE mode). By contrast, PCI DMA devices generally have more than enough address lines to access the full system physical address space in 32-bit processors. Therefore, every HAL provides mappings between the *logical address* ranges that DMA devices can access and *physical address* ranges of each computer.

Each adapter object is associated with one or more map registers, depending on the amount of data to be transferred and the amount of available memory. During DMA transfers, the HAL uses each map register to alias a device-accessible logical page to a page of physical memory in the CPU. In effect, map registers provide scatter/gather support for drivers that use DMA, regardless of whether their devices have scatter/gather capabilities.

The following figure illustrates such a physical-to-logical address mapping for the driver of an ISA DMA device that does not have scatter/gather capabilities.



The previous figure shows the following types of mappings:

1. Each map register maps a range of physical addresses (pointed to by solid lines) to low-order logical addresses (dotted lines) for an ISA DMA device.

   Here, three map registers are used to alias three paged ranges of data in system physical memory to three page-sized ranges of low-order logical addresses for an ISA DMA device.

2. The ISA device uses the mapped logical addresses to access system memory during DMA operations.

   For a comparable PCI DMA device, three map registers would also be used for three page-sized ranges of data. However, the mapped logical address ranges would not necessarily be identical to the corresponding physical address ranges, so a PCI device would also use logical addresses to access system memory.

3. Each entry in the MDL maps a location in virtual address space to a physical address.

Note the correspondence between a map register and a virtual-to-physical entry in the MDL:

- Each map register and each virtual entry in an MDL maps at most a full physical page of data for a DMA transfer operation.
- Each map register and each virtual entry in an MDL might map less than a full page of data. For example, the initial virtual entry in an MDL can map to an offset from the physical page boundary, as shown earlier in the **Physical, Logical, and Virtual Address Mappings** figure.
- Each map register and each virtual entry in an MDL maps, at a minimum, one byte.

In an IRP requesting a read or write operation, each virtual entry in the opaque-to-drivers MDL at **Irp->MdlAddress** represents a page boundary in the system physical memory for a user buffer. Similarly, each additional map register needed for a single DMA transfer represents a page boundary in the device-accessible logical address range aliased to system physical memory.

On every Windows platform, each adapter object has an associated set of one or more map registers located at a platform-specific (and opaque-to-drivers) base address. From a driver's point of view, the map register base shown in the figure illustrating address mapping for a sample ISA DMA device is a handle for a set of map registers that could be hardware registers in a chip, in a system DMA controller, or in a bus-master adapter, or could even be HAL-created virtual registers in system memory.

The number of map registers available with an adapter object can vary for different devices and Windows platforms. For example, the HAL can make more map registers available to drivers that use system DMA on some platforms than on other platforms because the DMA controllers on different Windows platforms have different capabilities.

Mapping Bus-Relative Addresses to Virtual Addresses

7/1/2019 • 2 minutes to read • Edit Online

Some processors implement separate memory and I/O address spaces, while other processors do not. Because of these differences in hardware platforms, the mechanism drivers use to access I/O- or memory-resident device resources differs from platform to platform.

A driver requests device I/O and memory resources in response to the PnP manager's **IRP_MN_QUERY_RESOURCE_REQUIREMENTS** IRP. Depending on the hardware architecture, the HAL can assign I/O resources in I/O space or in memory space, and can assign memory resources in I/O space or in memory space.

If the HAL uses bus-relative memory space to access device resources (such as device registers), a driver must map I/O space into virtual memory so that it can access these resources. The driver can determine whether the resources are I/O- or memory-resident by inspecting the translated resources passed to the driver by the PnP manager at device startup. If the HAL uses I/O space, no mapping is required.

Specifically, when a driver receives an **IRP_MN_START_DEVICE** request, it should examine the structures at **IrpSp->Parameters.StartDevice.AllocatedResources** and **IrpSp->Parameters.StartDevice.AllocatedResourcesTranslated**, which describe the raw (bus-relative) and translated resources, respectively, that the PnP manager has assigned to the device. Drivers should save a copy of each resource list in the device extension as an aid to debugging.

The resource lists are paired **CM_RESOURCE_LIST** structures, in which each element of the raw list corresponds to the same element of the translated list. For example, if **AllocatedResources.List**[0] describes a raw I/O port range, **AllocatedResourcesTranslated.List**[0] describes the same range after translation. Each translated resource includes a physical address and the type of the resource.

If a driver is assigned a translated memory resource (**CmResourceTypeMemory**), it must call **MmMapIoSpace** to map the physical address into a virtual address through which it can access device registers. For a driver to operate in a platform-independent manner, it should check every returned, translated resource and map it, if necessary.

A kernel-mode driver should take the following steps, in response to an IRP_MN_START_DEVICE request, to ensure access to all device resources

1. Copy **IrpSp->Parameters.StartDevice.AllocatedResources** to the device extension.
2. Copy **IrpSp->Parameters.StartDevice.AllocatedResourcesTranslated** to the device extension.
3. In a loop, inspect each descriptor element in **AllocatedResourcesTranslated**. If the descriptor resource type is **CmResourceTypeMemory**, call **MmMapIoSpace**, passing the physical address and length of the translated resource.

When the driver receives an **IRP_MN_STOP_DEVICE** or **IRP_MN_REMOVE_DEVICE** request from the PnP manager, it must release the mappings by calling **MmUnmapIoSpace** in a similar loop. The driver should also call **MmUnmapIoSpace** if it must fail the **IRP_MN_START_DEVICE** request.

The raw resource type indicates which HAL access routine a driver should call (**READ_REGISTER_*XXX***, **WRITE_REGISTER_*XXX***, **READ_PORT_*XXX***, **WRITE_PORT_*XXX***). Most drivers do not have to check the raw resource list to determine which of these routines to use, because the driver itself requested the resource or the driver writer knows the required type given the nature of the device hardware.

For a resource in I/O space (**CmResourceTypePort**, **CmResourceTypeInterrupt**, **CmResourceTypeDma**), the

driver should use the low-order 32 bits of the returned physical address to access the device resource, for example, through the HAL's read and write **READ_REGISTER_*XXX***, **WRITE_REGISTER_*XXX***, **READ_PORT_*XXX***, **WRITE_PORT_*XXX*** routines.

Using the Kernel Stack

6/25/2019 • 2 minutes to read • Edit Online

The size of the kernel-mode stack is limited to approximately three pages. Therefore, when passing data to internal routines, drivers cannot pass large amounts of data on the kernel stack.

To avoid running out of kernel-mode stack space, use the following design guidelines:

- Avoid making deeply nested calls from one internal driver routine to another, if each routine passes data on the kernel stack.
- Make sure that you limit the number of recursive calls that can occur, if you design a driver that has a recursive routine.

In other words, the call-tree structure of a driver should be relatively flat. You can call the **IoGetStackLimits** and **IoGetRemainingStackSize** routines to determine the kernel stack space that is available, or **KeExpandKernelStackAndCallout** to expand it. Note that the size of the kernel-mode stack can vary among different hardware platforms and different versions of the operating system.

Running out of kernel stack space causes a fatal system error. Therefore, it is better for a driver to allocate system-space memory than to run out of kernel stack space. However, nonpaged pool is also a limited system resource.

Generally, the kernel-mode stack resides in memory, however it can occasionally be paged out if the thread enters a wait state that specifies user mode. See **KeSetKernelStackSwapEnable** for information on how to temporarily disable kernel stack paging for the current thread. For performance reasons, it is not recommended to disable kernel stack paging globally, but if you want to do so during a debugging session, see Disable paging of kernel stacks

Because the kernel stack might be paged out, please be cautious about passing stack-based buffers (i.e. local variables) to DMA or any routine that runs at DISPATCH_LEVEL or above.

Using Lookaside Lists

6/25/2019 • 5 minutes to read • Edit Online

Drivers that must allocate fixed-size buffers dynamically to perform on-demand I/O operations can use the **Ex*Xxx*LookasideListEx** or **Ex*Xxx*LookasideList** support routines. After such a driver initializes its lookaside list, the operating system will hold some number of dynamically allocated buffers of the given size in the driver's lookaside list, effectively reserving a set of reusable, fixed-size buffers for the driver. The format and contents of a driver's fixed-size buffers (also known as *entries*) in its lookaside list are driver-determined.

For example, storage class drivers that must set up SCSI request blocks (SRBs) for the underlying SCSI port/miniport drivers use lookaside lists. Such a class driver allocates buffers for SRBs on an as-needed basis from its lookaside list and releases each SRB buffer back to the lookaside list for the lookaside list to reuse whenever an SRB is returned to the class driver in a completed IRP. Because a storage class driver cannot predetermine how many SRBs it has to use at any time because I/O demand on the driver increases and falls, a lookaside list is a convenient and economical way to manage the allocation and deallocation of buffers for fixed-size SRBs in such a driver.

The operating system maintains state about all paged and nonpaged lookaside lists that are currently being used, dynamically tracking the demand for allocations and deallocations of entries in all lists, and available system pool for new entries. When demand for allocations is high, the operating system increases the number of entries it holds in each lookaside list. When demand falls again, it frees surplus lookaside entries back to system pool.

Lookaside lists are thread-safe. A lookaside list has built-in synchronization to enable multiple, concurrently running threads in a driver to share a lookaside list. These threads can safely allocate buffers from the shared lookaside list and free these buffers to the list without requiring the driver to explicitly synchronize these operations. However, to avoid possible leaks and data corruption, a set of threads that share a lookaside list must explictly synchronize the initialization and deletion of the list.

Lookaside list interfaces

Starting with Windows Vista, the **LOOKASIDE_LIST_EX** structure describes a lookaside list that can contain either paged or nonpaged buffers. If a driver provides custom *Allocate* and *Free* routines for this lookaside list, these routines receive a private context as an input parameter. A driver can use this context to collect private data for the lookaside list. For example, the context might be used to count the number of list entries that are dynamically allocated and freed by the list. For a code example that shows how to use a context in this way, see **ExInitializeLookasideListEx**.

The following system-supplied routines support lookaside lists that are described by a **LOOKASIDE_LIST_EX** structure:

ExAllocateFromLookasideListEx

ExDeleteLookasideListEx

ExFlushLookasideListEx

ExFreeToLookasideListEx

ExInitializeLookasideListEx

Starting with Windows 2000, the **PAGED_LOOKASIDE_LIST** structure describes a lookaside list that contains paged buffers. If a driver provides custom *Allocate* and *Free* routines for this lookaside list, these routines do not receive a private context as an input parameter. For this reason, if your driver is intended to run only on

Windows Vista and later versions of Windows, consider using the **LOOKASIDE_LIST_EX** structure instead of the **PAGED_LOOKASIDE_LIST** structure for your lookaside lists. The following system-supplied routines support lookaside lists that are described by a **PAGED_LOOKASIDE_LIST** structure:

ExAllocateFromPagedLookasideList

ExDeletePagedLookasideList

ExFreeToPagedLookasideList

ExInitializePagedLookasideList

Starting with Windows 2000, the **NPAGED_LOOKASIDE_LIST** structure describes a lookaside list that contains nonpaged buffers. If a driver provides custom *Allocate* and *Free* routines for this lookaside list, these routines do not receive a private context as an input parameter. Again, if your driver is intended to run only on Windows Vista and later versions of Windows, consider using the **LOOKASIDE_LIST_EX** structure instead of the **NPAGED_LOOKASIDE_LIST** structure for your lookaside lists. The following system-supplied routines support lookaside lists that are described by an **NPAGED_LOOKASIDE_LIST** structure:

ExAllocateFromNPagedLookasideList

ExDeleteNPagedLookasideList

ExFreeToNPagedLookasideList

ExInitializeNPagedLookasideList

Implementation guidelines

To implement a lookaside list that uses a **LOOKASIDE_LIST_EX** structure, follow these design guidelines:

- Call **ExInitializeLookasideListEx** to set up a lookaside list. In this call, specify whether the entries in the lookaside list are to be paged or nonpaged buffers. Use nonpaged buffers if the driver itself or any underlying driver to which it passes its lookaside list entries might access these entries at IRQL >= DISPATCH_LEVEL. Use paged buffers only if accesses to the driver's lookaside list entries always occur at IRQL <= APC_LEVEL.
- The **LOOKASIDE_LIST_EX** structure for the lookaside list must always reside in nonpaged system memory regardless of whether the entries in the list are paged or nonpaged.
- For better performance, pass **NULL** pointers for the *Allocate* and *Free* parameters to **ExInitializeLookasideListEx** unless the allocation and deallocation routines must do more than merely allocate and free memory for lookaside list entries. For example, these routines might record information about the driver's usage of dynamically allocated buffers.
- A driver-supplied *Allocate* routine can pass the input parameters (*PoolType*, *Tag*, and *Size*) that it receives directly to the **ExAllocatePoolWithTag** or **ExAllocatePoolWithQuotaTag** routine to allocate a new buffer.
- For every call to **ExAllocateFromLookasideListEx**, make the reciprocal call to **ExFreeToLookasideListEx** as soon as possible whenever a previously allocated entry is no longer being used.

Supplying *Allocate* and *Free* routines that do nothing more than call **ExAllocatePoolWithTag** and **ExFreePool**, respectively, wastes processor cycles. **ExAllocateFromLookasideListEx** makes the necessary calls to **ExAllocatePoolWithTag** and **ExFreePool** automatically when a driver passes **NULL** *Allocate* and *Free* pointers to **ExInitializeLookasideListEx**.

Any driver-supplied *Allocate* routine must not allocate memory for an entry from paged pool to be held in a

nonpaged lookaside list or vice versa. It must also allocate fixed-size entries, because any subsequent driver call to **ExAllocateFromLookasideListEx** returns the first entry currently held in the lookaside list unless the list is empty. That is, a call to **ExAllocateFromLookasideListEx** causes a call to the driver-supplied *Allocate* routine only if the given lookaside list is currently empty. Therefore, at each call to **ExAllocateFromLookasideListEx**, the returned entry will be exactly the size that the driver needs only if all entries in the lookaside list are of a fixed size. A driver-supplied *Allocate* routine should also not change the *Tag* value that the driver originally passed to **ExInitializeLookasideListEx**, because changes in the pool-tag value would make debugging and tracking the driver's memory usage more difficult.

Calls to **ExFreeToLookasideListEx** store previously allocated entries in the lookaside list unless the list is already *full* (that is, the list contains the system-determined maximum number of entries). For better performance, a driver should make a reciprocal call to **ExFreeToLookasideListEx** as quickly as it can for every call that it makes to **ExAllocateFromLookasideListEx**. When a driver frees entries back to its lookaside list quickly, that driver's next call to **ExAllocateFromLookasideListEx** is far less likely to incur the performance penalty of dynamically allocating memory for a new entry.

Similar guidelines apply to a lookaside list that uses a **PAGED_LOOKASIDE_LIST** or **NPAGED_LOOKASIDE_LIST** structure.

Making Drivers Pageable

6/25/2019 • 2 minutes to read • Edit Online

By default, the linker assigns names such as ".text" and ".data" to the code and data sections of a driver image file. When the driver is loaded, the I/O manager makes these sections nonpaged. A nonpaged section is always memory-resident.

A driver developer has the option to make designated parts of a driver pageable so that Windows can move these parts to the paging file when they are not in use. To make a code or data section pageable, the driver developer assigns a name that begins with "PAGE" to the section. The I/O manager checks the names of the sections when it loads a driver. If a section name begins with "PAGE", the I/O manager makes the section pageable.

Code that runs at IRQL >= DISPATCH_LEVEL must be memory-resident. That is, this code must be either in a nonpageable segment, or in a pageable segment that is locked in memory. If code that is running at IRQL >= DISPATCH_LEVEL causes a page fault, a bug check occurs. Drivers can use the **PAGED_CODE** macro to verify that pageable functions are called only at appropriate IRQLs.

If a code or data section is pageable, the driver can lock the section in memory by calling the **MmLockPagableCodeSection** or **MmLockPagableDataSection** routine. The section remains locked until the driver calls the **MmUnlockPagableImageSection** routine to unlock it. While the pageable section is locked, it behaves the same as a nonpaged section.

For information about how to assign names to code and data sections, see **MmLockPagableCodeSection** and **MmLockPagableDataSection**.

This section includes the following topics:

When Should Code and Data Be Pageable?

6/25/2019 • 2 minutes to read • Edit Online

You can make all or part of your driver pageable. Paging driver code can reduce the size of the driver's load image, thus freeing system space for other uses. It is most practical for drivers of sporadically used devices, such as modems and CD-ROMs, or for parts of drivers that are rarely called.

Driver code that does any of the following must be memory-resident. That is, this code must be either in a nonpaged section, or in a paged section that is locked in memory when the code runs.

- Runs at or above IRQL = DISPATCH_LEVEL.
- Acquires spin locks.
- Calls any of the kernel's object support routines, such as **KeReleaseMutex** or **KeReleaseSemaphore**, in which the *Wait* parameter is set to **TRUE**. If the kernel is called with *Wait* set to **TRUE**, the call returns with IRQL at DISPATCH_LEVEL.

Driver code must be running at IRQL < DISPATCH_LEVEL when the code does anything that might cause a page fault. Code can cause a page fault if it does any of the following:

- Accesses paged pool that is not locked in memory.
- Calls a pageable routine.
- Accesses unlocked user buffers in the context of the user thread.

Typically, you should make a section paged if the total amount of all the pageable code (or data) is at least 4 kilobytes (KB). Whenever possible, you should isolate purely pageable code (or data) into a separate section from code (or data) that can sometimes be pageable but must sometimes be locked. For example, combining purely pageable code and locked-on-demand code causes more system space to be locked down for the combined section than is necessary. However, if a driver has less than 4 KB of possibly pageable code (or data), you might combine that code (or data) with locked-on-demand code (or data) into one section, saving system space.

Detecting Code That Can Be Pageable

6/25/2019 • 2 minutes to read • Edit Online

To detect code that runs at IRQL >= DISPATCH_LEVEL, use the **PAGED_CODE** macro. In debug mode, this macro generates a message if the code runs at IRQL >= DISPATCH_LEVEL. Add the macro as the first statement in a routine to mark the whole routine as paged code, as the following example shows:

```
NTSTATUS
MyDriverXxx(
    IN OUT PVOID ParseContext OPTIONAL,
    OUT PHANDLE Handle
    )
{
    NTSTATUS Status;

    PAGED_CODE();
.
.
.
}
```

To make sure that you are doing this correctly, run the Driver Verifier against your finished driver with the **Force IRQL Checking** option enabled. This option causes the system to automatically page out all pageable code every time that the driver raises IRQL to DISPATCH_LEVEL or above. Using the Driver Verifier, you can quickly find any driver bugs in this area. Otherwise, these bugs will typically be found only by customers and they can frequently be very hard for you to reproduce.

Isolating Pageable Code

6/25/2019 • 2 minutes to read • Edit Online

A routine that uses a spin lock cannot be paged. However, in some cases you can isolate the operations that require a spin lock in a separate routine that will not be included in the pageable section.

For example, consider the following fragment:

```
//  PAGED_CODE();

KeInitializeEvent( &event, NotificationEvent, FALSE );
irp = IoBuildDeviceIoControlRequest( IRP_MJ_DEVICE_CONTROL,
                                     DeviceObject,
                                     (PVOID) NULL,
                                     0,
                                     (PVOID) NULL,
                                     0,
                                     FALSE,
                                     &event,
                                     &ioStatus );
if (irp) {
   irpSp = IoGetNextIrpStackLocation( irp );
   irpSp->MajorFunction = IRP_MJ_FILE_SYSTEM_CONTROL;
   irpSp->MinorFunction = IRP_MN_LOAD_FILE_SYSTEM;
   status = IoCallDriver( DeviceObject, irp );
   if (status == STATUS_PENDING) {
   (VOID) KeWaitForSingleObject( &event,
                                 Executive,
                                 KernelMode,
                                 FALSE,
                                 (PLARGE_INTEGER) NULL );
   }
}

SPINLOCKUSE !
KeAcquireSpinLock( &IopDatabaseLock, &irql );
// Code inside spin lock ...

DeviceObject->ReferenceCount--;

if (!DeviceObject->ReferenceCount && !DeviceObject->AttachedDevice) {
   //Unload the driver
   .
   .
   .
} else {
   KeReleaseSpinLock( &IopDatabaseLock, irql );
}
```

The preceding routine could be made pageable (saving about 160 bytes) by moving the few lines of code that reference a spin lock into a separate routine.

In addition, remember that driver code must not be marked as pageable if it calls any **Ke*Xxx*** support routines, such as **KeReleaseMutex** or **KeReleaseSemaphore**, in which the *Wait* parameter is set to **TRUE**. Such a call returns with IRQL at DISPATCH_LEVEL.

Locking Pageable Code or Data

7/9/2019 • 7 minutes to read • Edit Online

Certain kernel-mode drivers, such as the serial and parallel drivers, do not have to be memory-resident unless the devices they manage are open. However, as long as there is an active connection or port, some part of the driver code that manages that port must be resident to service the device. When the port or connection is not being used, the driver code is not required. In contrast, a driver for a disk that contains system code, application code, or the system paging file must always be memory-resident because the driver constantly transfers data between its device and the system.

A driver for a sporadically used device (such as a modem) can free system space when the device it manages is not active. If you place in a single section the code that must be resident to service an active device, and if your driver locks the code in memory while the device is being used, this section can be designated as pageable. When the driver's device is opened, the operating system brings the pageable section into memory and the driver locks it there until no longer needed.

The system CD audio driver code uses this technique. Code for the driver is grouped into pageable sections according to the manufacturer of CD device. Certain brands might never be present on a given system. Also, even if a CD-ROM exists on a system, it might be accessed infrequently, so grouping code into pageable sections by CD type makes sure that code for devices that do not exist on a particular computer will never be loaded. However, when the device is accessed, the system loads the code for the appropriate CD device. Then the driver calls the **MmLockPagableCodeSection** routine, as described below, to lock its code into memory while its device is being used.

To isolate the pageable code into a named section, mark it with the following compiler directive:

#pragma alloc_text(PAGE*Xxx, **RoutineName*)

The name of a pageable code section must start with the four letters "PAGE" and can be followed by up to four characters (represented here as ***Xxx***) to uniquely identify the section. The first four letters of the section name (that is, "PAGE") must be capitalized. The *RoutineName* identifies an entry point to be included in the pageable section.

The shortest valid name for a pageable code section in a driver file is simply PAGE. For example, the pragma directive in the following code example identifies `RdrCreateConnection` as an entry point in a pageable code section named PAGE.

```
#ifdef  ALLOC_PRAGMA
#pragma alloc_text(PAGE, RdrCreateConnection)
#endif
```

To make pageable driver code resident and locked in memory, a driver calls **MmLockPagableCodeSection**, passing an address (typically the entry point of a driver routine) that is in the pageable code section. **MmLockPagableCodeSection** locks in the whole contents of the section that contains the routine referenced in the call. In other words, this call makes every routine associated with the same PAGE*Xxx* identifier resident and locked in memory.

MmLockPagableCodeSection returns a handle to be used when unlocking the section (by calling the **MmUnlockPagableImageSection** routine) or when the driver must lock the section from additional locations in its code.

A driver can also treat seldom-used data as pageable so that it, too, can be paged out until the device it supports is active. For example, the system mixer driver uses pageable data. The mixer device has no asynchronous I/O

associated with it, so this driver can make its data pageable.

The name of a pageable data section must start with the four letters "PAGE" and can be followed by up to four characters to uniquely identify the section. The first four letters of the section name (that is, "PAGE") must be capitalized.

Avoid assigning identical names to code and data sections. To make source code more readable, driver developers typically assign the name PAGE to the pageable code section because this name is short and it might appear in numerous alloc_text pragma directives. Longer names are then assigned to any pageable data sections (for example, PAGEDATA for data_seg, PAGEBSS for bss_seg, and so on) that the driver might require.

For example, the first two pragma directives in the following code example define two pageable data sections, PAGEDATA and PAGEBSS. PAGEDATA is declared using the data_seg pragma directive and contains initialized data. PAGEBSS is declared using the bss_seg pragma directive and contains uninitialized data.

```
#pragma data_seg("PAGEDATA")
#pragma bss_seg("PAGEBSS")

INT Variable1 = 1;
INT Variable2;

CHAR Array1[64*1024] = { 0 };
CHAR Array2[64*1024];

#pragma data_seg()
#pragma bss_seg()
```

In this code example, `Variable1` and `Array1` are explicitly initialized and are therefore placed in the PAGEDATA section. `Variable2` and `Array2` are implicitly zero-initialized and are placed in the PAGEBSS section.

Implicitly initializing global variables to zero reduces the size of the on-disk executable file and is preferred over explicit initialization to zero. Explicit zero-initialization should be avoided except in cases where it is required in order to place a variable in a specific data section.

To make a data section memory-resident and lock it in memory, a driver calls **MmLockPagableDataSection**, passing a data item that appears in the pageable data section. **MmLockPagableDataSection** returns a handle to be used in subsequent locking or unlocking requests.

To restore a locked section's pageable status, call **MmUnlockPagableImageSection**, passing the handle value returned by **MmLockPagableCodeSection** or **MmLockPagableDataSection**, as appropriate. A driver's *Unload* routine must call **MmUnlockPagableImageSection** to release each handle it has obtained for lockable code and data sections.

Locking a section is an expensive operation because the memory manager must search its loaded module list before locking the pages into memory. If a driver locks a section from many locations in its code, it should use the more efficient **MmLockPagableSectionByHandle** after its initial call to **MmLockPagable*Xxx*Section**.

The handle passed to **MmLockPagableSectionByHandle** is the handle returned by the earlier call to **MmLockPagableCodeSection** or **MmLockPagableDataSection**.

The memory manager maintains a count for each section handle and increments this count every time that a driver calls **MmLockPagable*Xxx*** for that section. A call to **MmUnlockPagableImageSection** decrements the count. While the counter for any section handle is nonzero, that section remains locked in memory.

The handle to a section is valid as long as its driver is loaded. Therefore, a driver should call **MmLockPagable*Xxx*Section** only one time. If the driver requires additional locking calls, it should use **MmLockPagableSectionByHandle**.

If a driver calls any **MmLockPagable*Xxx*** routine for a section that is already locked, the memory manager

increments the reference count for the section. If the section is paged out when the lock routine is called, the memory manager pages in the section and sets its reference count to one.

Using this technique minimizes the driver's effect on system resources. When the driver runs, it can lock into memory the code and data that must be resident. When there are no outstanding I/O requests for its device, (that is, when the device is closed or if the device was never opened), the driver can unlock the same code or data, making it available to be paged out.

However, after a driver has connected interrupts, any driver code that can be called during interrupt processing always must be memory resident. While some device drivers can be made pageable or locked into memory on demand, some core set of such a driver's code and data must be permanently resident in system space.

Consider the following implementation guidelines for locking a code or data section.

- The primary use of the **Mm(Un)Lock*Xxx*** routines is to enable normally nonpaged code or data to be made pageable and brought in as nonpaged code or data. Drivers such as the serial driver and the parallel driver are good examples: if there are no open handles to a device such a driver manages, parts of code are not needed and can remain paged out. The redirector and server are also good examples of drivers that can use this technique. When there are no active connections, both of these components can be paged out.
- The whole pageable section is locked into memory.
- One section for code and one for data per driver is efficient. Many named, pageable sections are generally inefficient.
- Keep purely pageable sections and paged but locked-on-demand sections separate.
- Remember that **MmLockPagableCodeSection** and **MmLockPagableDataSection** should not be frequently called. These routines can cause heavy I/O activity when the memory manager loads the section. If a driver must lock a section from several locations in its code, it should use **MmLockPagableSectionByHandle**.

Paging an Entire Driver

6/25/2019 • 2 minutes to read • Edit Online

A driver that uses the **MmLockPagable*Xxx*** support routines and specifies paged and discardable sections consists of nonpaged sections, paged sections, and an INIT section that is discarded after driver initialization.

After a device driver connects interrupts for the devices it manages, the driver's interrupt handling path must be resident in system space. The interrupt-handling code must be part of the driver section that cannot be paged out, in case an interrupt occurs.

Two additional memory manager routines, **MmPageEntireDriver** and **MmResetDriverPaging**, can be used to override the pageable or nonpageable attributes of all sections that make up a driver image. These routines enable a driver to be paged out in its entirety when the device it manages is not being used and cannot generate interrupts.

Examples of system drivers that are completely pageable are the win32k.sys driver, the serial driver, the mailslot driver, the beep driver and the null driver.

A serial driver is typically used intermittently. Until a port it manages is opened, a serial driver can be paged out completely. As soon as a port is opened, the parts of the serial driver that must be memory-resident must be brought into nonpaged system space. Other parts of the driver can remain pageable.

A driver that can be completely paged out should call **MmPageEntireDriver** during driver initialization before interrupts are connected.

When a device managed by a paged-out driver receives an open request, the driver is paged in. Then, the driver must call **MmResetDriverPaging** before it connects to interrupts. Calling **MmResetDriverPaging** causes the memory manager to treat the driver's sections according to the attributes acquired during compilation and linkage. Any section that is nonpaged, such as a text section, will be paged into nonpaged system memory; pageable sections will be paged in as they are referenced.

Such a driver must keep a reference count of open handles to its devices. The driver increments the count at each open request for any device and decrements the count at each close request. When the count reaches zero, the driver should disconnect interrupts and then call **MmPageEntireDriver**. If a driver manages more than one device, the count must be zero for all such devices before the driver can call **MmPageEntireDriver**.

It is the driver's responsibility to do whatever synchronization is necessary when changing the reference count, and to prevent the reference count from changing while the pageable state of the driver is changing. That is, in SMP computers, the driver must make sure that **MmPageEntireDriver** cannot be in progress on one processor, while on another processor, an open call is causing interrupts to be connected and the reference count to be incremented.

Accessing Read-Only System Memory

6/25/2019 • 2 minutes to read • Edit Online

The Windows memory manager enforces read-only access of pages that are not marked as writable.

Read-only memory has always been protected in user mode. However, in Windows NT 4.0 and earlier versions, read-only memory was not protected in kernel mode.

If a Windows kernel-mode driver or application tries to write to a read-only memory segment, the system issues a bug check. For more information, see **Bug Check 0xBE: ATTEMPTED_WRITE_TO_READONLY_MEMORY**.

Intercepting System Calls

Some drivers intercept system calls by overwriting the driver's own code and inserting jump instructions or other changes. Because the driver's own code is read-only, this technique will cause a bug check to be issued.

Global Strings

If a global string is to be modified, it must not be declared as a pointer to a constant value:

```
CHAR *myString = "This string cannot be modified.";
```

In this case, the linker might put the string in a read-only memory segment. Then an attempt to modify the string will result in a bug check.

Instead, the string should be explicitly declared as an array of L-value characters:

```
CHAR myString[] = "This string can be modified.";
```

This declaration makes sure that the string is put in writable memory.

Accessing User-Space Memory

6/25/2019 • 2 minutes to read • Edit Online

A driver cannot directly access memory through user-mode virtual addresses unless it is running in the context of the user-mode thread that caused the driver's current I/O operation and it is using that thread's virtual addresses.

Only highest-level drivers, such as FSDs, can be sure their dispatch routines will be called in the context of such a user-mode thread. A highest-level driver can call **MmProbeAndLockPages** to lock down a user buffer before setting up an IRP for lower drivers.

Lowest-level and intermediate drivers that set up their device objects for buffered I/O or direct I/O can rely on the I/O manager or a highest-level driver to pass valid access to locked-down user buffers or to system-space buffers in IRPs.

No-Execute (NX) Nonpaged Pool

12/5/2018 • 2 minutes to read • Edit Online

As a best practice, drivers for Windows 8 and later versions of Windows should allocate most or all of their nonpaged memory from the no-execute (NX) nonpaged pool. By allocating memory from NX nonpaged pool, a kernel-mode driver improves security by preventing malicious software from executing instructions in this memory.

Starting with Windows 8, kernel-mode drivers can allocate memory from a pool of NX nonpaged memory. This pool is managed by a general-purpose, kernel-mode memory allocator that operates similarly to the user-mode Win32 heap allocator. The memory in this pool is NX and nonpaged. The x86, x64, and ARM processor architectures enable memory pages to be designated as NX to prevent the execution of instructions in these pages. Typically, a kernel-mode driver uses memory allocated from nonpaged pool to store data, and does not require the ability to execute instructions in this memory.

Support for Legacy Drivers

In Windows 7 and earlier versions of Windows, all memory allocated from the nonpaged pool is executable. To encourage porting of these drivers to use NX nonpaged pool in Windows 8 and later versions of Windows, Microsoft provides several opt-in mechanisms to enable developers to update their drivers with minimal effort. For more information, see NX Pool Opt-In Mechanisms.

For backward compatibility, driver binaries that run on Windows 7 and earlier versions of Windows, and that allocate memory from the executable nonpaged pool, will run on Windows 8 and later versions of Windows without modification. However, these drivers do not take advantage of the improved security of the NX nonpaged pool.

NX and Execute Pool Types

6/25/2019 • 2 minutes to read • Edit Online

To indicate whether memory allocated from a nonpaged pool should be no-execute (NX), you can use two new pool types starting with Windows 8. These pool types are designated by the following **POOL_TYPE** enumeration values:

NonPagedPoolNx
NX nonpaged pool. Instructions cannot be executed in memory allocated from this pool.

NonPagedPoolExecute
Executable nonpaged pool. Instruction execution is enabled in memory allocated from this pool.

Most drivers use allocated memory only to store data, and do not execute instructions in this memory. If you build your driver to run on Windows 8 and later versions of Windows, allocate **NonPagedPoolNx** memory from the NX nonpaged pool whenever possible. Only drivers that need to execute instructions from nonpaged memory should allocate **NonPagedPoolExecute** memory from the executable nonpaged pool.

For existing drivers that are built for Windows 7 and earlier versions of Windows, when you use the **NonPagedPool** pool type your driver allocates memory from the executable pool. The **NonPagedPool** constant name has the same value as the **NonPagedPoolExecute** constant name that is defined starting with Windows 8. Therefore, these constants refer to the same pool type.

If a driver written for Windows 7 or an earlier version of Windows is built for Windows 8 or a later version of Windows, instances of the **NonPagedPool** pool type should be replaced by either **NonPagedPoolNx** or **NonPagedPoolExecute**. The driver developer either can explicitly perform this replacement, or can implicitly perform the replacement by using one of the opt-in mechanisms that is provided to aid developers in porting their drivers to Windows 8. For more information, see NX Pool Opt-In Mechanisms.

NX Pool Compatibility Issues

6/25/2019 • 3 minutes to read • Edit Online

When you use the NX nonpaged pool in driver binaries for Windows 8, you will find compatibility issues if you run these binaries on earlier versions of Windows.

Windows 8 is the first version of Windows to support the NX nonpaged pool. However, a large number of existing kernel-mode driver binaries are available for Windows 7 and earlier versions of Windows that run on the x86, x64, and IA64 processor architectures. To allocate nonpaged memory, these drivers use the executable nonpaged pool instead the NX nonpaged pool. For backward compatibility, kernel-mode driver binaries that run on Windows 7, and on some earlier versions of Windows, and that allocate memory from the nonpaged pool, will run on Windows 8 without modification. However, these drivers do not take advantage of the availability of NX nonpaged pool in Windows 8.

Running Existing Driver Binaries on Windows 8

A driver binary that is built for Windows 7 (or possibly for an earlier version of Windows), and that uses the **NonPagedPool** pool type, is not prevented from running on Windows 8. To enable backward compatibility, the **NonPagedPoolExecute** constant is defined to have the same value as the **NonPagedPool** constant in the **POOL_TYPE** enumeration. Thus, in any version of Windows in which this driver runs, the memory that the driver allocates from nonpaged pool is always executable.

Windows 8 is the first version of Windows to support the ARM architecture. Thus, there are no driver binaries for ARM that are built for earlier versions of Windows and that require backward compatibility. Instead, all drivers written for Windows on ARM are expected to specify **NonPagedPoolNx** instead of **NonPagedPoolExecute** in their nonpaged pool allocations unless they explicitly require executable memory.

If a driver is ported to ARM from x86, x64, or IA64, the POOL_NX_OPTIN_AUTO opt-in mechanism is automatically applied during the driver build process. This opt-in mechanism uses the preprocessor to replace, by default, all instances of the **NonPagedPool** constant name with **NonPagedPoolNx**. If necessary, you can use the POOL_NX_OPTOUT opt-out mechanism to overrride this opt-in mechanism on a per-file basis.

Other Compatibility Issues

The **NonPagedPoolNx** pool type is supported starting with Windows 8. Do not use this pool type in drivers for earlier versions of Windows. The pool allocators in these earlier versions of Windows do not operate correctly when the driver requests an allocation with a **NonPagedPoolNx** pool type.

In versions of Windows before Windows 8, the **NonPagedPoolExecute** pool type can be freely used as a substitute for the **NonPagedPool** pool type. The **POOL_TYPE** enumeration defines **NonPagedPool** and **NonPagedPoolExecute** to have the same value.

NX Pool Type Porting Guidelines

When you port your driver code to Windows 8 or later from an earlier version of Windows, there are several ways to add support for the **NonPagedPoolNx** and **NonPagedPoolExecute** pool types. From the following list, choose the approach that best fits your requirements:

- If your driver is not intended to run on a version of Windows earlier than Windows 8, replace most or all instances of **NonPagedPool** with **NonPagedPoolNx**. Only rarely should the developer replace an instance of **NonPagedPool** with **NonPagedPoolExecute.**

- If your driver source code targets both Windows 8 and earlier versions of Windows, and you ship a different driver binary for each version, use the POOL_NX_OPTIN_AUTO opt-in mechanism. This approach does not require replacing the instances of **NonPagedPool** in the driver source. For more information, see NX Pool Opt-In Mechanisms.
- If your driver source code targets both Windows 8 and earlier versions of Windows, and you ship a single driver binary to run on all supported versions, use the POOL_NX_OPTIN opt-in mechanism. This approach does not require replacing the instances of **NonPagedPool** in the driver source. For more information, see NX Pool Opt-In Mechanisms.

By using one of these three approaches, most drivers can be ported quickly and with little effort.

Avoid simply replacing all instances of **NonPagedPool** in your driver code with **NonPagedPoolExecute**. Use the **NonPagedPoolExecute** pool type only for pool allocations that must be executable (for example, to run code produced by a just-in-time, or JIT, compiler).

NX Pool Opt-In Mechanisms

6/25/2019 • 2 minutes to read • Edit Online

To port kernel-mode driver code to Windows 8 from earlier versions of Windows, you should use the **NonPagedPoolNx** type of memory pool as a best practice. You can use one of several porting aids to easily "opt in" to use the **NonPagedPoolNx** pool type by default.

These porting aids use one or both of the following techniques to enable the driver to use NX nonpaged pool:

- Use a `#define` preprocessor statement to create a globally defined macro name.
- Call an inline function from the **DriverEntry** routine.

For most kernel-mode driver code, these porting aids enable developers to update their drivers with minimal effort.

In this section

TOPIC	DESCRIPTION
Single Binary Opt-In: POOL_NX_OPTIN	To build a single driver binary that runs both in Windows 8 and in earlier versions of Windows, use the POOL_NX_OPTIN opt-in mechanism. This is a porting aid for third-party hardware vendors who supply a single driver binary to support multiple Windows versions.
Multiple Binary Opt-In: POOL_NX_OPTIN_AUTO	If you are a hardware vendor who supplies different driver binaries for different versions of Windows, you can use the POOL_NX_OPTIN_AUTO opt-in mechanism. This porting aid builds a separate driver binary for Windows 8 and for each earlier version of Windows that your driver supports.
Selective Opt-Out: POOL_NX_OPTOUT	You can globally enable one of the no-execute (NX) pool opt-in mechanisms for a set of driver source files, and then override this opt-in mechanism for one or more selected source files with POOL_NX_OPTOUT. This allows the selected source files to continue to use executable nonpaged memory. You can use the POOL_NX_OPTOUT opt-out mechanism with either the POOL_NX_OPTIN or the POOL_NX_OPTIN_AUTO opt-in mechanism.

Single Binary Opt-In: POOL_NX_OPTIN

6/25/2019 • 2 minutes to read • Edit Online

To build a single driver binary that runs both in Windows 8 and in earlier versions of Windows, use the POOL_NX_OPTIN opt-in mechanism. This is a porting aid for third-party hardware vendors who supply a single driver binary to support multiple Windows versions.

To use this opt-in mechanism, do the following:

- Define POOL_NX_OPTIN = 1 for all source files that you want to opt-in. To do this, include the following preprocessor definition in the appropriate property page for your driver project:

  ```
  C_DEFINES=$(C_DEFINES) -DPOOL_NX_OPTIN=1
  ```

- In your **DriverEntry** (or equivalent) routine, include the following function call:

  ```
  ExInitializeDriverRuntime(DrvRtPoolNxOptIn);
  ```

  This call must occur before the driver makes any allocations that use the **NonPagedPool** pool type or makes any calls to the **ExInitializeNPagedLookasideList** routine. **ExInitializeDriverRuntime** is a force inline function and can be called on Windows 8 or later versions of Windows.

For most drivers, these two tasks are sufficient to enable the opt-in mechanism for the single driver binary.

Implementation details

POOL_NX_OPTIN works by replacing **NonPagedPool** with a global **POOL_TYPE** variable, `ExDefaultNonPagedPoolType`, which is initialized either to **NonPagedPoolNx** (for Windows 8 and later versions of Windows) or to **NonPagedPoolExecute** (for earlier versions of Windows). This opt-in mechanism enables your kernel-mode driver to run both on Windows 8, with the enhanced protection of NX pool, and on earlier versions of Windows, which do not support NX pool. The macro that converts instances of the **NonPagedPool** constant name to **NonPagedPoolNx** also converts instances of **NonPagedPoolCacheAligned** to **NonPagedPoolNxCacheAligned**.

Support for static libraries (.lib projects)

You can use the POOL_NX_OPTIN opt-in mechanism for a .lib project, but projects that link to the .lib generally must also use POOL_NX_OPTIN. At a minimum, the project that implements the **DriverEntry** routine must contain the following function call:

```
ExInitializeDriverRuntime(DrvRtPoolNxOptIn);
```

Multiple Binary Opt-In: POOL_NX_OPTIN_AUTO

12/5/2018 • 2 minutes to read • Edit Online

If you are a hardware vendor who supplies different driver binaries for different versions of Windows, you can use the POOL_NX_OPTIN_AUTO opt-in mechanism. This porting aid builds a separate driver binary for Windows 8 and for each earlier version of Windows that your driver supports.

To use this opt-in mechanism, define POOL_NX_OPTIN_AUTO=1 for all source files that you want to opt-in. To do this, include the following preprocessor definition in the appropriate property page for your driver project:

```
C_DEFINES=$(C_DEFINES) -DPOOL_NX_OPTIN_AUTO=1
```

For most drivers, this definition is sufficient to enable the opt-in mechanism to create a different binary for each version of Windows that you support.

Implementation details

The POOL_NX_OPTIN_AUTO definition redefines the **NonPagedPool** constant name to **NonPagedPoolNx**. The redefined pool type is still a compile-time constant. The macro that converts instances of the **NonPagedPool** constant name to **NonPagedPoolNx** also converts instances of **NonPagedPoolCacheAligned** to **NonPagedPoolNxCacheAligned.**

Selective Opt-Out: POOL_NX_OPTOUT

12/5/2018 • 2 minutes to read • Edit Online

You can globally enable one of the no-execute (NX) pool opt-in mechanisms for a set of driver source files, and then override this opt-in mechanism for one or more selected source files with POOL_NX_OPTOUT. This allows the selected source files to continue to use executable nonpaged memory. You can use the POOL_NX_OPTOUT opt-out mechanism with either the POOL_NX_OPTIN or the POOL_NX_OPTIN_AUTO opt-in mechanism. For more information, see NX Pool Opt-In Mechanisms.

To use the POOL_NX_OUTPUT opt-out mechanism to override the opt-in mechanism in a selected source file, add the following definition to this file:

```
#define POOL_NX_OPTOUT 1
```

This definition overrides the global opt-in settings in the selected file, and prevents instances of the **NonPagedPool** constant name from being replaced. Insert this definition into the file before the first instance of **NonPagedPool** in the file.

An alternative to using the POOL_NX_OPTOUT opt-out mechanism in a source file is to explicitly replace each instance of **NonPagedPool** in the file with **NonPagedPoolExecute**.

Section Objects and Views

12/5/2018 • 2 minutes to read • Edit Online

A *section object* represents a section of memory that can be shared. A process can use a section object to share parts of its memory address space (memory sections) with other processes. Section objects also provide the mechanism by which a process can map a file into its memory address space.

Each memory section has one or more corresponding *views*. A view of a section is a part of the section that is actually visible to a process. The act of creating a view for a section is known as *mapping* a view of the section. Each process that is manipulating the contents of a section has its own view; a process can also have multiple views (to the same or different sections).

This section contains the following topics:

File-Backed and Page-File-Backed Sections

12/5/2018 • 2 minutes to read • Edit Online

All memory sections are supported ("backed") by disk files that can contain, either temporarily or permanently, the data to be shared. When you create a section, you can identify a specific data file to which the section will be backed. Such sections are called *file-backed* sections. If you do not identify a backing file, the section is backed by the system's paging file and the section is called a *page-file-backed* section. The data in file-backed sections can be permanently written to disk. Data in page-file-backed sections is never permanently written to disk.

A *file-backed* section reflects the contents of an actual file on disk; in other words, it is a memory-mapped file. Any access to memory locations within a given file-backed section corresponds to accesses to locations in the associated file. If a process maps the view as read-only, any data that is read from the view is transparently read from the file. Similarly, if the process maps the view as read/write, any data that is read from the view or written to the view is transparently read from or written to the file. In either case, the view's virtual memory does not use any space in the page files. A file-backed section can also be mapped as copy-on-write. In that case, the view's data is read from the file, but any data written to the view is not written to the file; instead it is discarded after the final view is unmapped and the last handle to the section is closed.

A page-file-backed section is backed by the page files instead of by any explicit file on the disk. Any changes that are made to a page-file-backed section are automatically discarded after the section object is destroyed. Page-file-backed sections can be used as shared memory segments between two processes.

Any section, file-backed or not, can be shared between two processes. The same physical memory address range is mapped to a virtual memory address range within each process (though not necessarily to the same virtual address).

Managing Memory Sections

6/25/2019 • 2 minutes to read • Edit Online

A driver can create a section object by calling **ZwCreateSection**, which returns a handle to the section object. Use the *FileHandle* parameter to specify the backing file, or **NULL** if the section is not file-backed. Additional handles to the section object can be opened by using **ZwOpenSection**.

To make the data that belongs to a section object accessible within the current process' address space, a view of the section must be mapped. Drivers can map a view of a section into the current process' address space by using the **ZwMapViewOfSection** routine. The *SectionOffset* parameter specifies the byte offset where the view begins within the section, and the *ViewSize* specifies the number of bytes to be mapped.

The *Protect* parameter specifies the allowed operations on the view. Specify PAGE_READONLY for a read-only view, PAGE_READWRITE for a read/write view, and PAGE_WRITECOPY for a copy-on-write view.

No physical memory is allocated for a view until the virtual memory range is accessed. The first access of the memory range causes a page fault; the system then allocates a page to hold that memory location. If the section is file-backed, the system reads the contents of the file that corresponds to that page and copies it into memory. (Note that unused section objects and views do use some paged and nonpaged pool for bookkeeping purposes.)

After a driver is no longer using a view, it unmaps it by making a call to **ZwUnmapViewOfSection**. After the driver is no longer using the section object, it closes the section handle with **ZwClose**. Note that after the view is mapped and no other views are going to be mapped, it is safe to immediately call **ZwClose** on the section handle; the view (and section object) continue to exist until the view is unmapped. This is the recommended practice because it reduces the risk of the driver failing to close the handle.

Security Issues for Section Objects and Views

6/25/2019 • 2 minutes to read • Edit Online

Drivers that create sections and views that are not to be shared with user mode must use the following protocol when they are working with sections and views:

- The driver must use a kernel handle when it is opening a handle to the section object. Drivers can make sure that a handle is a kernel handle by either creating it in the system process, or specifying the OBJ_KERNEL_HANDLE attribute for the handle. For more information, see Object Handles.
- The view must be mapped only from a system thread. (Otherwise, the view is accessible from the process whose context it is created in.) A driver can make sure that the view is mapped from the system process by using a system worker thread to perform the mapping operation. For more information, see System Worker Threads and Driver Thread Context.

Drivers that share a view with a user-mode process must use the following protocol when they are working with sections and views:

- The driver, not the user-mode process, must create the section object and map the views.
- As mentioned earlier, the driver must use a kernel handle when it is opening a handle to the section object. Drivers can make sure that a handle is a kernel handle by either creating it in the system process, or specifying the OBJ_KERNEL_HANDLE attribute for the handle. For more information, see Object Handles.
- The view is mapped in the thread context of the process that shares the view. A highest-level driver can guarantee the view is mapped in the current process context by performing the mapping operation in a dispatch routine, such *DispatchDeviceControl*. Dispatch routines of lower-level drivers run in an arbitrary thread context, and thus cannot safely map a view in a dispatch routine. For more information, see Driver Thread Context.
- All memory accesses to the view within the driver must be protected by **try**-**except** blocks. A malicious user-mode application could unmap the view or change the protection state of the view. Either would cause a system crash unless protected by a **try**-**except** block. For more information, see Handling Exceptions.

The driver must also validate the contents of the view as necessary. The driver writer cannot assume that only a trusted user-mode component will have access to the view.

A driver that must share a section object with a user-mode application (that must be able to create its own views) must use the following protocol:

- The driver, not the user-mode process, must create the section object. Drivers must never use a handle that was passed from user mode.
- Before passing the handle to user mode, the driver must call **ObReferenceObjectByHandle** to obtain a reference to the section object. This prevents a malicious application from deleting the section object by closing the handle. The object reference should be stored in the driver's device extension.
- After the driver is no longer using the section object, it must call **ObDereferenceObject** to release the object reference.

On systems that run Microsoft Windows Server 2003 with Service Pack 1 (SP1) and later versions, only kernel-mode drivers can open **\Device\PhysicalMemory**. However, drivers can decide to give a handle to a user application. To prevent security issues, only user applications that the driver trusts should be given access to **\Device\PhysicalMemory**.

Using MDLs

10/17/2019 • 5 minutes to read • Edit Online

An I/O buffer that spans a range of contiguous virtual memory addresses can be spread over several physical pages, and these pages can be discontiguous. The operating system uses a *memory descriptor list* (MDL) to describe the physical page layout for a virtual memory buffer.

An MDL consists of an **MDL** structure that is followed by an array of data that describes the physical memory in which the I/O buffer resides. The size of an MDL varies according to the characteristics of the I/O buffer that the MDL describes. System routines are available to calculate the required size of an MDL and to allocate and free the MDL.

An MDL structure is semi-opaque. Your driver should directly access only the **Next** and **MdlFlags** members of this structure. For a code example that uses these two members, see the following Example section.

The remaining members of an MDL are opaque. Do not access the opaque members of an MDL directly. Instead, use the following macros, which the operating system provides to perform basic operations on the structure:

MmGetMdlVirtualAddress returns the virtual memory address of the I/O buffer that is described by the MDL.

MmGetMdlByteCount returns the size, in bytes, of the I/O buffer.

MmGetMdlByteOffset returns the offset within a physical page of the beginning of the I/O buffer.

You can allocate an MDL with the **IoAllocateMdl** routine. To free the MDL, use the **IoFreeMdl** routine. Alternatively, you can allocate a block of nonpaged memory and then format this block of memory as an MDL by calling the **MmInitializeMdl** routine.

Neither **IoAllocateMdl** nor **MmInitializeMdl** initializes the data array that immediately follows the MDL structure. For an MDL that resides in a driver-allocated block of nonpaged memory, use **MmBuildMdlForNonPagedPool** to initialize this array to describe the physical memory in which the I/O buffer resides.

For pageable memory, the correspondence between virtual and physical memory is temporary, so the data array that follows the MDL structure is valid only under certain circumstances. Call **MmProbeAndLockPages** to lock the pageable memory into place and to initialize this data array for the current layout. The memory will not be paged out until the caller uses the **MmUnlockPages** routine, at which point the contents of the data array are no longer valid.

The **MmGetSystemAddressForMdlSafe** routine maps the physical pages that are described by the specified MDL to a virtual address in system address space, if they are not already mapped to system address space. This virtual address is useful for drivers that might have to look at the pages to perform I/O, because the original virtual address might be a user address that can be used only in its original context and can be deleted at any time.

Note that when you build a partial MDL by using the **IoBuildPartialMdl** routine, the caller should use **MmGetMdlVirtualAddress** instead of the **MmGetSystemAddressForMdlSafe** routine when determining the virtual address to pass in. **IoBuildPartialMdl** uses the address that **MmGetMdlVirtualAddress** returns from the source MDL to determine the offset for the target MDL. If the addresses are different (for example, when the first address is a user address), passing the address that **MmGetSystemAddressForMdlSafe** returns can cause data corruption or a bug check.

When a driver calls **IoAllocateMdl**, it can associate an IRP with the newly allocated MDL by specifying a pointer to the IRP as the *Irp* parameter of **IoAllocateMdl**. An IRP can have one or more MDLs associated with it. If the IRP has a single MDL associated with it, the IRP's **MdlAddress** member points to that MDL. If the IRP has

multiple MDLs associated with it, **MdlAddress** points to the first MDL in a linked list of MDLs that are associated with the IRP, known as an *MDL chain*. The MDLs are linked by their **Next** members. The **Next** member of the last MDL in the chain is set to **NULL**.

If, when the driver calls **IoAllocateMdl**, it specifies **FALSE** for the *SecondaryBuffer* parameter, the IRP's **MdlAddress** member is set to point to the new MDL. If *SecondaryBuffer* is **TRUE**, the routine inserts the new MDL at the end of the MDL chain.

When the IRP is completed, the system unlocks and frees all the MDLs that are associated with the IRP. The system unlocks the MDLs before it queues the I/O completion routine and frees them after the I/O completion routine executes.

Drivers can traverse the MDL chain by using the **Next** member of each MDL to access the next MDL in the chain. Drivers can manually insert MDLs into the chain by updating the **Next** members.

MDL chains are typically used to manage an array of buffers that are associated with a single I/O request. (For example, a network driver could use one buffer for each IP packet in a network operation.) Each buffer in the array has its own MDL in the chain. When the driver completes the request, it combines the buffers into a single large buffer. The system then automatically cleans up all the allocated MDLs for the request.

The I/O manager is a frequent source of I/O requests. When the I/O manager completes an I/O request, the I/O manager frees the IRP and frees any MDLs that are attached to the IRP. Some of these MDLs might have been attached to the IRP by drivers that are located beneath the I/O manager in the device stack. Similarly, if your driver is the source of an I/O request, your driver must clean up the IRP and any MDLs that are attached to the IRP when the I/O request completes.

Example

The following code example is a driver-implemented function that frees an MDL chain from an IRP:

```
VOID MyFreeMdl(PMDL Mdl)
{
    PMDL currentMdl, nextMdl;

    for (currentMdl = Mdl; currentMdl != NULL; currentMdl = nextMdl)
    {
        nextMdl = currentMdl->Next;
        if (currentMdl->MdlFlags & MDL_PAGES_LOCKED)
        {
            MmUnlockPages(currentMdl);
        }
        IoFreeMdl(currentMdl);
    }
}
```

If the physical pages that are described by an MDL in the chain are locked, the example function calls the **MmUnlockPages** routine to unlock the pages before it calls **IoFreeMdl** to free the MDL. However, the example function does not need to explicitly unmap the pages before it calls **IoFreeMdl**. Instead, **IoFreeMdl** automatically unmaps the pages when it frees the MDL.

Controlling Device Access

6/25/2019 • 2 minutes to read • Edit Online

Access to a device is controlled by a security descriptor (and the ACL it contains). A security descriptor for a device object can be specified when the device object is created, or set in the registry.

Controlling Device Access for WDM Drivers

When a WDM driver (other than certain bus drivers) creates a device object, the Plug and Play manager determines a security descriptor for the device. The order of operations is as follows.

1. The PnP manager calls the driver's *AddDevice* routine.
2. The driver's *AddDevice* routine calls **IoCreateDevice** to create the device object and attach it to the device object stack.
3. The PnP manager updates the security descriptor for the newly-created device object.

For a WDM driver, the PnP manager determines the security descriptor for the device object as follows.

1. If the device has a security descriptor setting in the registry, it is applied to every object in the device stack.
2. Otherwise, if the device's setup class has a security descriptor setting in the registry, it is applied to every object in the device stack.
3. Otherwise, the PnP manager leaves the default security descriptor for each object unchanged. In this case, the default security descriptor for the stack is determined by the device type and device characteristics of the PDO.

For most device types and characteristics, the default security descriptor gives full access (GENERIC_ALL) to administrators, and read, write, and execute access (GENERIC_READ | GENERIC_WRITE | GENERIC_EXECUTE) access to everyone else.

For more information about how to set a security descriptor for a device or device setup class in the registry, see Setting Device Object Properties in the Registry.

If a device is operated in raw mode, then the PnP manager cannot determine a security descriptor for the device object. In that case, the bus driver must provide a security descriptor; see below.

Controlling Device Access for WDM Bus Drivers

A WDM bus driver must provide a security descriptor for the PDO of every device that can be operated in raw mode. Use **IoCreateDeviceSecure** to create the device object with a security descriptor.

If the bus driver does not operate a device in raw mode, then it is not required to supply a security descriptor. The PnP manager determines the security descriptor, as described above. The bus driver can supply a security descriptor if it must ensure that its PDOs have stricter security settings than the default descriptor. Any descriptor specified by the bus driver is overridden by settings in the registry.

For more information about creating device objects, see Creating a Device Object.

Controlling Device Access for Non-WDM Drivers

Non-WDM drivers must specify a default security descriptor and class GUID for any named device objects they create.

Use the **IoCreateDeviceSecure** routine to create the named device object and to specify the default security descriptor and class GUID for that device. The security descriptor is specified in a subset of SDDL. For more

information, see SDDL for Device Objects.

The system overrides the default security descriptor with any security settings in the registry for the specified class GUID. The driver must specify its own unique GUID for the device. Use the GuidGen tool to generate a unique GUID. (GuidGen is included in the Microsoft Windows SDK.)

Controlling Device Namespace Access

6/25/2019 • 2 minutes to read • Edit Online

In the Windows Driver Model (WDM), every device object has an associated *namespace*. Names in the device's namespace are paths that begin with the device's name. For a device named "*Device**DeviceName*", its namespace consists of any name of the form "*Device**DeviceName**FileName*". (For a file system, *FileName* is an actual name of a file on the file system.)

A WDM driver receives open requests for all names in the device's namespace. The driver treats an open request for "*Device**DeviceName*" as an open of the device object itself. If the driver implements support for open requests into the device's namespace, then it treats an open request for "*Device**DeviceName**FileName*" as an open of a "file" within the device object's namespace (where the notion of "file" for the device is driver-determined).

Most drivers do not implement support for open operations into the device's namespace, but all drivers must provide security checks to prevent unauthorized access to the device's namespace. By default, security checks for file open requests within the device's namespace, (for example, "*Device**DeviceName**FileName*") are left entirely up to the driver—the device object ACL is not checked by the operating system.

If a device object's FILE_DEVICE_SECURE_OPEN characteristic is set, the system applies the device object's security descriptor to all file open requests in the device's namespace. Drivers can set FILE_DEVICE_SECURE_OPEN when they create the device object with **IoCreateDevice** or **IoCreateDeviceSecure**. For WDM drivers, FILE_DEVICE_SECURE_OPEN can also be set in the registry. It can also be set in the registry for device objects of non-WDM drivers that are created by **IoCreateDeviceSecure**. For more information about setting device object properties, such as the device characteristics, in the registry, see Setting Device Object Properties in the Registry. For more information about device characteristics, see Specifying Device Characteristics.

Drivers for devices that do not support namespaces must use one of two methods to ensure that file open requests within the device's namespace are handled correctly:

- The driver's device objects have the FILE_DEVICE_SECURE_OPEN device characteristic set. The driver can then treat any open request into the device's namespace as an open request for the device object.
- The driver can fail any **IRP_MJ_CREATE** requests that specify an **IrpSp->FileObject->FileName** parameter whose length is nonzero. In this case, open requests for the device are subject to the system's ACL check, while all file open requests within the device's namespace are failed by the driver. (Drivers that support exclusive opens must use this option.)

Drivers for devices that do support namespaces can also use two methods to secure file open requests into the device's namespace:

- The driver's device objects have the FILE_DEVICE_SECURE_OPEN device characteristic set. This ensures that the security settings for the device apply uniformly to the device's namespace. (The driver is responsible for implementing support for the namespace in its *DRIVER_DISPATCH* callback function.)
- The driver checks any ACLs for the file name in its *DispatchCreate* routine. (Even in this case the driver should set the FILE_DEVICE_SECURE_OPEN characteristic unless opens into the device's namespace can have weaker security settings than the device object.)

The FILE_DEVICE_SECURE_OPEN characteristic is checked at the top of the stack, so filter device objects must copy the **Characteristics** member of the next-lower device object after attaching.

SDDL for Device Objects

6/25/2019 • 6 minutes to read • Edit Online

The Security Descriptor Definition Language (SDDL) is used to represent security descriptors. Security for device objects can be specified by an SDDL string that is placed in an INF file or passed to **IoCreateDeviceSecure**. The Security Descriptor Definition Language is fully documented in the Microsoft Windows SDK documentation.

While INF files support the full range of SDDL, only a subset of the language is supported by the **IoCreateDeviceSecure** routine. This subset is defined here.

SDDL strings for device objects are of the form "D:P" followed by one or more expressions of the form "(A;;*Access*;;;*SID*)". The *SID* value specifies a security identifier that determines to whom the *Access* value applies (for example, a user or group). The *Access* value specifies the access rights allowed for the SID. The *Access* and *SID* values are as follows.

Note When using SDDL for device objects, your driver must link against Wdmsec.lib.

Access
Specifies an **ACCESS_MASK** value that determines the allowed access. This value can be written either as a hexadecimal value in the form "0x*hex*", or as a sequence of two-letter symbolic codes that represent access rights.

The following codes can be used to specify generic access rights.

CODE	GENERIC ACCESS RIGHT
GA	GENERIC_ALL
GR	GENERIC_READ
GW	GENERIC_WRITE
GX	GENERIC_EXECUTE

The following codes can be used to specify specific access rights.

CODE	SPECIFIC ACCESS RIGHT
RC	READ_CONTROL
SD	DELETE
WD	WRITE_DAC
WO	WRITE_OWNER

Note that GENERIC_ALL grants all the rights listed in the above two tables, including the ability to change the

ACL.

SID
Specifies the SID that is granted the specified access. SIDs represent accounts, aliases, groups, users, or computers.

The following SIDs represent *accounts* on the machine.

SID	DESCRIPTION
SY	System Represents the operating system itself, including its user-mode components.
LS	Local Service A predefined account for local services (which also belongs to Authenticated and World). This SID is available starting with Windows XP.
NS	Network Service A predefined account for network services (which also belongs to Authenticated and World). This SID is available starting with Windows XP.

The following SIDs represent *groups* on the machine.

SID	DESCRIPTION
BA	Administrators The built-in Administrators group on the machine.
BU	Built-in User Group Group covering all local user accounts, and users on the domain.
BG	Built-in Guest Group Group covering users logging in using the local or domain guest account.

The following SIDs describe the extent to which a user has been authenticated.

SID	DESCRIPTION
AU	Authenticated Users Any user recognized by the local machine or by a domain. Note that users logged in using the Builtin Guest account are not authenticated. However, members of the Guests group with individual accounts on the machine or the domain are authenticated.

SID	DESCRIPTION
AN	Anonymous Logged-on User Any user logged on without an identity, such as an anonymous network session. Note that users logging in using the Builtin Guest account are neither authenticated nor anonymous. This SID is available starting with Windows XP.

The following SIDs describe how the user logged into the machine.

SID	DESCRIPTION
IU	Interactive Users Users who initially logged onto the machine "interactively", such as local logons and Remote Desktops logons.
NU	Network Logon User Users accessing the machine remotely, without interactive desktop access (for example, file sharing or RPC calls).
WD	World Before Windows XP, this SID covered every session, whether authenticated users, anonymous users, or the Builtin Guest account. Starting with Windows XP, this SID does not cover anonymous logon sessions; it covers only authenticated users and the Builtin Guest account. Note that untrusted or "restricted" code is also not covered by the World SID. For more information, see the description of the Restricted Code (RC) SID in the following table.

The following SIDs deserve special mention.

SID	DESCRIPTION

SID	DESCRIPTION
RC	Restricted Code This SID is used to control access by untrusted code. ACL validation against tokens with RC consists of two checks, one against the token's normal list of SIDs (containing WD for instance), and one against a second list (typically containing RC and a subset of the original token SIDs). Access is granted only if a token passes both tests. As such, RC actually works in combination with other SIDs. Any ACL that specifies RC must also specify WD. When RC is paired with WD in an ACL, a superset of Everyone including untrusted code is described. Untrusted code might be code launched using the Run As option in Explorer. By default, World does not cover untrusted code.
UD	User-Mode Drivers This SID grants access to user-mode drivers. Currently, this SID covers only drivers that are written for the User-Mode Driver Framework (UMDF). This SID is available starting with Windows 8. In earlier versions of Windows, which do not recognize the "UD" abbreviation, you must specify the fully qualified form of this SID (S-1-5-84-0-0-0-0-0) to grant access to UMDF drivers. For more information, see Controlling Device Access in the User-Mode Driver Framework documentation.

SDDL Examples For Device Objects

This section describes the predefined SDDL strings found in Wdmsec.h. You can also use these as templates to define new SDDL strings for device objects.

SDDL_DEVOBJ_KERNEL_ONLY

"D:P"

SDDL_DEVOBJ_KERNEL_ONLY is the "empty" ACL. User-mode code (including processes running as system) cannot open the device.

A PnP bus driver could use this descriptor when creating a PDO. The INF file could then specify looser security settings for the device. By specifying this descriptor, the bus driver would ensure that no attempt to open the device before the INF was processed would succeed.

Similarly, a non-WDM driver could use this descriptor to make its device objects inaccessible until the appropriate user-mode program (such as an installer) sets the final security descriptor in the registry.

In all of these cases, the default is tight security, loosened as necessary.

SDDL_DEVOBJ_SYS_ALL

"D:P(A;;GA;;;SY)"

SDDL_DEVOBJ_SYS_ALL is similar to SDDL_DEVOBJ_KERNEL_ONLY, except that in addition to kernel-mode code, user-mode code running as System is also allowed to open the device for any access.

A legacy driver might use this ACL to start with tight security settings, and let its service open the device up at run

time to individual users by using the **SetFileSecurity** user-mode function. In this case, the service would have to be running as System.

SDDL_DEVOBJ_SYS_ALL_ADM_ALL

"D:P(A;;GA;;;SY)(A;;GA;;;BA)"

SDDL_DEVOBJ_SYS_ALL_ADM_ALL allows the kernel, system, and administrator complete control over the device. No other users may access the device.

SDDL_DEVOBJ_SYS_ALL_ADM_RWX_WORLD_R

"D:P(A;;GA;;;SY)(A;;GRGWGX;;;BA)(A;;GR;;;WD)"

SDDL_DEVOBJ_SYS_ALL_ADM_RWX_WORLD_R allows the kernel and system complete control over the device. By default the administrator can access the entire device, but cannot change the ACL (the administrator must take control of the device first.)

Everyone (the World SID) is given read access. Untrusted code cannot access the device (untrusted code might be code launched using the Run As option in Explorer. By default, World does not cover Restricted code.)

Also note that traversal access is not granted to normal users. As such, this might not be an appropriate descriptor for a device with a namespace.

SDDL_DEVOBJ_SYS_ALL_ADM_RWX_WORLD_R_RES_R

"D:P(A;;GA;;;SY)(A;;GRGWGX;;;BA)(A;;GR;;;WD)(A;;GR;;;RC)"

SDDL_DEVOBJ_SYS_ALL_ADM_RWX_WORLD_R_RES_R allows the kernel and system complete control over the device. By default the administrator can access the entire device, but cannot change the ACL (the administrator must take control of the device first.)

Everyone (the World SID) is given read access. In addition, untrusted code is also allowed to access code. Untrusted code might be code launched using the Run As option in Explorer. By default, World does not cover Restricted code.

Also note that traversal access is not granted to normal users. As such, this might not be an appropriate descriptor for a device with a namespace.

Note that the above SDDL strings do not include any inheritance modifiers. As such, they are only appropriate for device objects and should not be used for files or registry keys. For more information about specifying inheritance using SDDL, see the Microsoft Windows SDK documentation.

Access Rights

6/25/2019 • 2 minutes to read • Edit Online

An *access right* is the right to perform a particular operation on the object. For example, the FILE_READ_DATA access right specifies the right to read from a file.

When you open a handle to an object, you specify a set of access rights corresponding to the operations that may be performed on the object. The system checks the specified access rights against the object's security descriptor to see if each operation is permitted for the current user. (For more information, see Security Descriptors.)

Access rights come in two types:

A *specific* access right is a right to perform a single operation. Specific access rights can depend on the type of object.

A *generic* access right is a right to perform one of a set of similar operations. Generic access rights are independent of the type of object.

Standard access rights are specific access rights that apply to all types of objects. For example, the DELETE access right is the right to delete an object, regardless of type. For more information about the available standard access rights, see **ACCESS_MASK**.

Objects also have specific access rights that depend on the type of the object. For example, the FILE_READ_DATA represents the right to read from a file, while the KEY_QUERY_VALUE represents the right to read the value entries for a registry key.

An object type can have zero, one, or more access rights that correspond to the general notion of reading from or writing to an object. For example, in addition to FILE_READ_DATA, file objects have the FILE_READ_ATTRIBUTES access right, which represents to read a file's metadata (such as file creation time). Key objects have both KEY_QUERY_VALUE and KEY_ENUMERATE_SUBKEYS, which represents the right to read the subkeys of a key.

To simplify specifying all access rights that correspond to a general notion such as reading or writing, the system provides generic access rights. The system maps a generic access right to the appropriate set of specific access rights for the object.

The system provides the following generic access rights:

- GENERIC_READ
- GENERIC_WRITE
- GENERIC_EXECUTE
- GENERIC_ALL

Thus, the system maps GENERIC_READ to a set of rights that includes FILE_READ_DATA and FILE_READ_ATTRIBUTES for a file, and KEY_QUERY_VALUE and KEY_ENUMERATE_SUBKEYS for a key. For more information about each generic access right, see **ACCESS_MASK**.

Security Descriptors

6/25/2019 • 2 minutes to read • Edit Online

Every object has a *security descriptor*, which contains the security settings for an object. In kernel-mode, the opaque **SECURITY_DESCRIPTOR** data type represents a security descriptor.

Information in a security descriptor is stored in *access control lists* (ACLs). An access control list is made up of a series of *access control entries* (ACEs).

A security descriptor has two separate ACLs:

- A *system ACL* (SACL), which determines which operations on an object are logged.
- A *discretionary ACL* (DACL), which determines which users can perform particular operations on the object.

Typically, a driver developer is only concerned with discretionary ACLs. For more information about system ACLs, see the Microsoft Windows SDK.

For a discretionary ACL, each ACE contains three pieces of information:

- A *security identifier* (SID). The security identifier determines who the ACE applies to. A SID can represent a single user, or a group of users. For example, the World SID represents the set of all users.
- A set of access rights. For a description of access rights, see Access Rights.
- Whether the set of access rights is granted, or denied.

For a driver, the most important security descriptors are those for the driver's device objects. For more information, see Securing Device Objects.

For more information about security descriptors in general, see the Windows SDK.

Privileges

6/25/2019 • 2 minutes to read • Edit Online

A *privilege* is a right that is associated with a process, rather than an object. A typical example of a privilege is **SeBackupPrivilege**, which confers on a process the right to back up files on a disk.

A few routines check the privilege of the current process before completing an operation. If a driver routine is executed by the system process, then the operation always succeeds, but if the driver routine is executed by a user process that does not have the required privilege, then the operation can fail.

The following table lists some examples of privileges and routines that can require them to succeed.

PRIVILEGE	ROUTINE THAT CAN REQUIRE PRIVILEGE
SeManageVolumePrivilege	**ZwSetInformationFile** with *FileInformationClass* = **FileValidDataLengthInformation**
SeTakeOwnershipPrivilege	**SeAccessCheck**
SeSecurityPrivilege	**SeAccessCheck**

Most system routines do not perform any privilege checks.

Handling IRPs

12/5/2018 • 2 minutes to read • Edit Online

This section describes how kernel-mode drivers handle *I/O request packets* (IRPs). It contains the following sections:

Overview of the Windows I/O Model

12/5/2018 • 2 minutes to read • Edit Online

Every operating system has an implicit or explicit I/O model for handling the flow of data to and from peripheral devices. One feature of the Microsoft Windows I/O model is its support for asynchronous I/O. In addition, the I/O model has the following general features:

- The I/O manager presents a consistent interface to all kernel-mode drivers, including lowest-level, intermediate, and file system drivers. All I/O requests to drivers are sent as I/O request packets (IRPs).
- I/O operations are layered. The I/O manager exports I/O system services, which user-mode protected subsystems call to carry out I/O operations on behalf of their applications and/or end users. The I/O manager intercepts these calls, sets up one or more IRPs, and routes them through possibly layered drivers to physical devices.
- The I/O manager defines a set of standard routines, some required and others optional, that drivers can support. All drivers follow a relatively consistent implementation model, given the differences among peripheral devices and the differing functionality required of bus, function, filter, and file system drivers.
- Like the operating system itself, drivers are object-based. Drivers, their devices, and system hardware are represented as objects. The I/O manager and other operating system components export kernel-mode support routines that drivers can call to get work done by manipulating the appropriate objects.

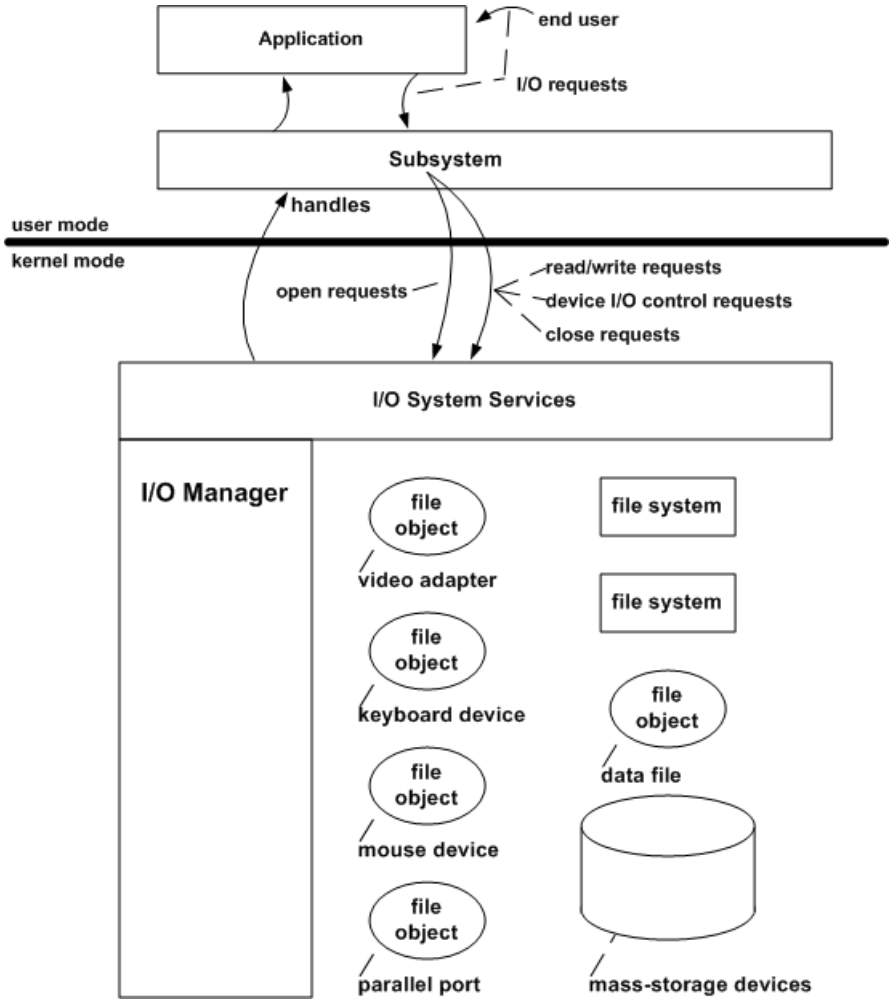
In addition to using IRPs to convey traditional I/O requests, the I/O manager works with the PnP and power managers to send IRPs containing PnP and power requests.

End-User I/O Requests and File Objects

12/5/2018 • 2 minutes to read • Edit Online

Kernel-mode drivers are hidden from end users by a protected subsystem that implements an already familiar programming interface, such as Windows or POSIX. Devices are visible to user-mode code, which includes protected subsystems, only as named file objects controlled by the I/O manager.

The following figure illustrates this relationship between an end user, a subsystem, and the I/O manager.



A protected subsystem, such as the Win32 subsystem, passes I/O requests to the appropriate kernel-mode driver through the I/O system services. The subsystem shown in the previous figure depends on support from the display, video adapter, keyboard, and mouse device drivers.

A protected subsystem insulates its end users and applications from having to know anything about kernel-mode components, including drivers. In turn, the I/O manager insulates protected subsystems from having to know anything about machine-specific device configurations or about drivers' implementations.

The I/O manager's layered approach also insulates most drivers from having to know anything about the following:

- Whether an I/O request originated in any particular protected subsystem, such as Win32 or POSIX
- Whether a given protected subsystem has particular kinds of user-mode drivers
- What any protected subsystem's I/O model and interface to drivers is

The I/O manager supplies drivers with a single I/O model, a set of kernel-mode support routines that drivers can use to carry out I/O operations, and a consistent interface between the originator of an I/O request and the drivers that must respond to it.

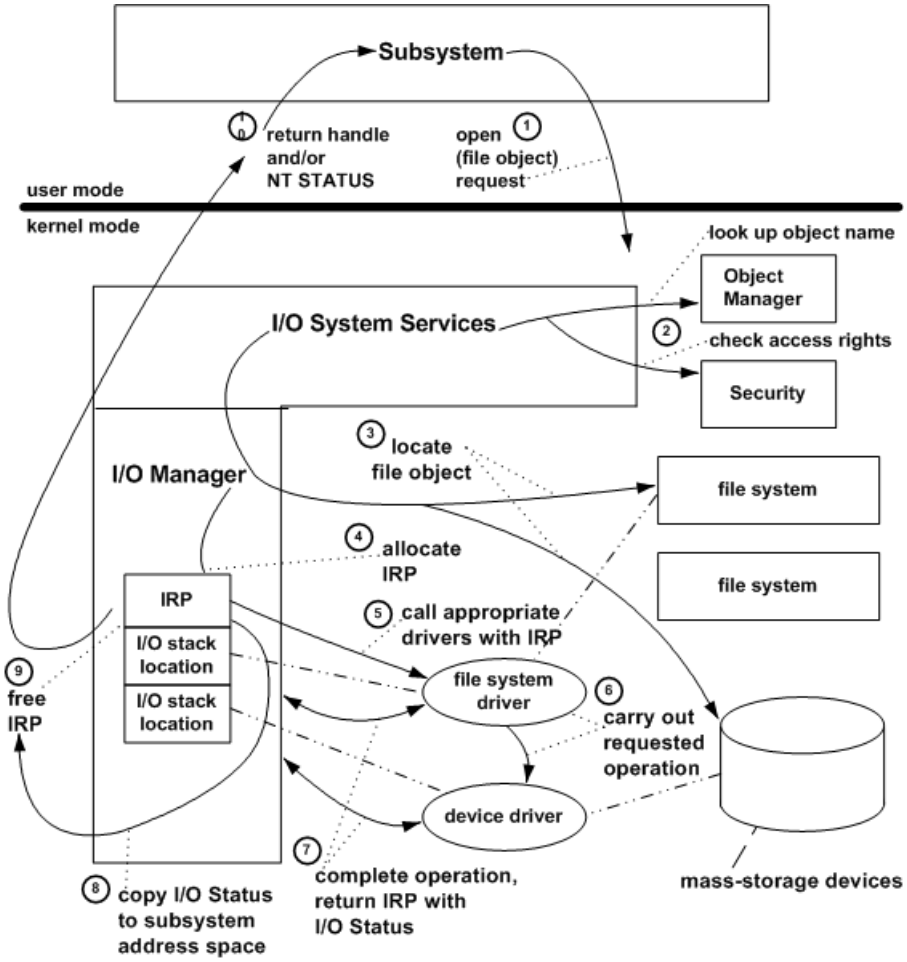
As shown in the previous figure, a subsystem and its native applications can access a driver's device or a file on a mass-storage device only through file object handles supplied by the I/O manager. To open such a file object or to obtain a handle for I/O to a device or a data file, a subsystem calls the I/O system services with a request to open a named file. The named file can have a subsystem-specific alias (symbolic link) to the kernel-mode name for the file object.

The I/O manager, which exports these system services, is then responsible for locating or creating the file object that represents the device or data file and for locating the appropriate driver(s).

Example I/O Request - An Overview

12/5/2018 • 2 minutes to read • Edit Online

The following figure shows an overview of what happens when a subsystem opens a file object representing a data file on behalf of an application.



1. The subsystem calls an I/O system service to open a named file.
2. The I/O manager calls the object manager to look up the named file and to help it resolve any symbolic links for the file object. It also calls the security reference monitor to check that the subsystem has the correct access rights to open that file object.
3. If the volume is not yet mounted, the I/O manager suspends the open request temporarily and calls one or more file systems until one of them recognizes the file object as something it has stored on one of the mass-storage devices the file system uses. When the file system has mounted the volume, the I/O manager resumes the request.
4. The I/O manager allocates memory for and initializes an IRP for the open request. To drivers, an open is equivalent to a "create" request.
5. The I/O manager calls the file system driver, passing it the IRP. The file system driver accesses its I/O stack location in the IRP to determine what operation it must carry out, checks parameters, determines if the requested file is in cache, and, if not, sets up the next-lower driver's I/O stack location in the IRP.
6. Both drivers process the IRP and complete the requested I/O operation, calling kernel-mode support routines supplied by the I/O manager and by other system components (not shown in the previous figure).

7. The drivers return the IRP to the I/O manager with the I/O status block set in the IRP to indicate whether the requested operation succeeded or why it failed.
8. The I/O manager gets the I/O status from the IRP, so it can return status information through the protected subsystem to the original caller.
9. The I/O manager frees the completed IRP.
10. The I/O manager returns a handle for the file object to the subsystem if the open operation was successful. If there was an error, it returns appropriate status to the subsystem.

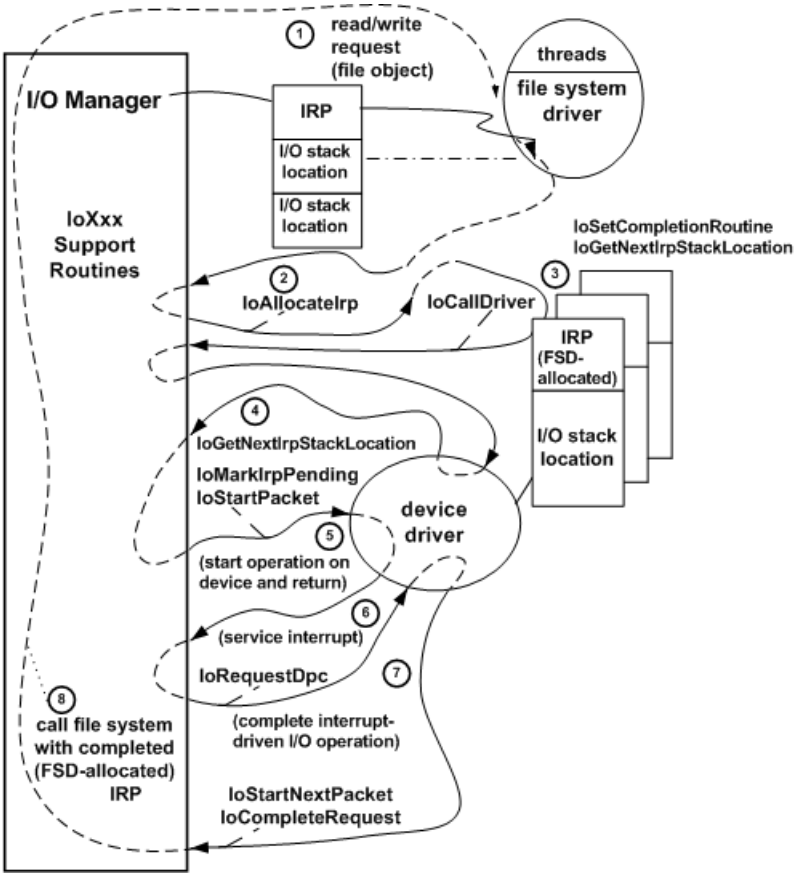
After a subsystem successfully opens a file object that represents a data file, a device, or a volume, the subsystem uses the returned handle to identify the file object in subsequent requests for device I/O operations (usually read, write, or device I/O control requests). To make such a request, the subsystem calls I/O system services. The I/O manager routes these requests as IRPs sent to appropriate drivers.

Example I/O Request - The Details

6/25/2019 • 6 minutes to read • Edit Online

The figure illustrating opening a file object shows an IRP with two I/O stack locations, but an IRP can have any number of I/O stack locations, depending on how many layered drivers will handle a given request.

The following figure illustrates in more detail how the drivers in the Opening a File Object figure use I/O support routines (**IoXxx** routines) to process the IRP for a read or write request.



1. The I/O manager calls the file system driver (FSD) with the IRP it has allocated for the subsystem's read/write request. The FSD accesses its I/O stack location in the IRP to determine what operation it should carry out.

2. The FSD can break the original request into smaller requests (possibly for more than one device driver) by calling an I/O support routine (**IoAllocateIrp**) one or more times to allocate additional IRPs. The additional IRPs are returned to the FSD with zero-filled I/O stack location(s) for lower-level driver(s). At its discretion, the FSD can reuse the original IRP, rather than allocating additional IRPs as shown in the previous figure, by setting up the next-lower driver's I/O stack location in the original IRP and passing it on to lower drivers.

3. For each driver-allocated IRP, the FSD in the previous figure calls an I/O support routine to register an FSD-supplied completion routine; in the completion routine, the FSD can determine whether lower drivers satisfied the request and can free each driver-allocated IRP when lower drivers have completed it. The I/O manager will call the FSD-supplied completion routine whether each driver-allocated IRP was completed successfully, completed with an error status, or canceled. A higher-level driver is responsible for freeing any IRPs it allocates and sets up on its own behalf for lower-level drivers. The I/O manager frees the IRPs that it allocates after all drivers have completed them.

   Next, the FSD calls an I/O support routine (**IoGetNextIrpStackLocation**) to access the next-lower-level

driver's I/O stack location in order to set up the request for the next-lower driver. (In the previous figure, the next lower driver happens to be the lowest-level driver.) The FSD then calls an I/O support routine (**IoCallDriver**) to pass that IRP on to the next-lower driver.

4. When it is called with the IRP, the lowest-level driver checks its I/O stack location to determine what operation (indicated by the **IRP_MJ_*XXX*** function code) it should carry out on the target device. The target device is represented by the device object in its designated I/O stack location and is passed with the IRP to the driver. The lowest-level driver can assume that the I/O manager has routed the IRP to an entry point that the driver defined for the **IRP_MJ_*XXX*** operation (here **IRP_MJ_READ** or **IRP_MJ_WRITE**) and that the higher-level driver has checked the validity of other parameters for the request.

   If there were no higher-level driver, the lowest-level driver would check whether the input parameters for an **IRP_MJ_*XXX*** operation are valid. If they are, the driver usually calls I/O support routines to tell the I/O manager that a device operation is pending on the IRP and to either queue the IRP or pass it on to another driver-supplied routine that accesses the target device (here, a physical or logical device: the disk or a partition on the disk).

5. The I/O manager determines whether the driver is already busy processing another IRP for the target device, queues the IRP if it is, and returns. Otherwise, the I/O manager routes the IRP to a driver-supplied routine that starts the I/O operation on its device. (At this stage, both drivers in the previous figure and the I/O manager return control.)

6. When the device interrupts, the driver's interrupt service routine (ISR) does only as much work as it must to stop the device from interrupting and to save necessary context about the operation. The ISR then calls an I/O support routine (**IoRequestDpc**) with the IRP to queue a driver-supplied DPC (Deferred Procedure Call) routine to complete the requested operation at a lower hardware priority than the ISR.

7. When the driver's DPC gets control, it uses the context (passed in the ISR's call to **IoRequestDpc**) to complete the I/O operation. The DPC calls a support routine to dequeue the next IRP (if any) and to pass that IRP on to the driver-supplied routine that starts I/O operations on the device (see Step 5). The DPC then sets status about the just-completed operation in the IRP's I/O status block and returns it to the I/O manager with **IoCompleteRequest**.

8. The I/O manager zeros the lowest-level driver's I/O stack location in the IRP and calls the file system's registered completion routine (see Step 3) with the FSD-allocated IRP. This completion routine checks the I/O status block to determine whether to retry the request or to update any internal state maintained about the original request and to free its driver-allocated IRP. The file system can collect status information for all driver-allocated IRPs it sends to lower-level drivers so that it can set I/O status and complete the original IRP. When the file system has completed the original IRP, the I/O manager returns and NTSTATUS value to the original requester (the subsystem's native function) of the I/O operation.

Like the file system driver shown in the Processing IRPs in Layered Drivers figure, any new driver that is added to a chain of existing drivers can do all of the following:

- Set its own completion routine into an IRP. The *IoCompletion* routine checks the I/O status block to determine whether lower drivers completed the IRP successfully, canceled the IRP, and/or completed it with an error. The completion routine can also update any IRP-specific state the driver might have saved, release any operation-specific resources the driver might have allocated, and so forth, before completing the IRP. In addition, the completion routine can postpone IRP completion (by informing the I/O manager that more processing is required on the IRP), and can send another request to the next-lower-level driver before allowing the IRP to complete.

- Set up the next-lower-level driver's I/O stack location in the IRPs it allocates and send requests to the next-lower-level driver.

- Pass any incoming requests on to lower drivers by setting up the next-lower driver's I/O stack location in

each IRP and calling **IoCallDriver**. (Note that for IRPs with major function code **IRP_MJ_POWER**, drivers must use **PoCallDriver**.)

Each driver-created device object represents a physical, logical, or virtual device for which a particular driver carries out I/O requests. For detailed information about creating and setting up a device object, see Device Objects and Device Stacks.

As the Processing IRPs in Layered Drivers figure also shows, most drivers process each IRP in stages through a driver-supplied set of system-defined *standard routines*, but drivers at different levels in a chain necessarily have different standard routines. For example, only lowest-level drivers handle interrupts from a physical device, so only a lowest-level driver would have an ISR and a DPC that completes interrupt-driven I/O operations. On the other hand, because such a driver knows that I/O is complete when it receives an interrupt from its device, it has no need for a completion routine. Only a higher-level driver would have one or more completion routines like the FSD in this figure.

Driver Thread Context

12/5/2018 • 2 minutes to read • Edit Online

As shown in the Processing IRPs in Layered Drivers figure, a file system is a two-part driver:

1. A file system driver (FSD), which executes in the context of a user-mode thread that calls an I/O system service

   The I/O manager sends the corresponding IRP to the FSD. If the FSD sets up a completion routine for an IRP, its completion routine is not necessarily called in the original user-mode thread's context.

2. A set of file system threads, and possibly an *FSP (file system process)*

   An FSD can create a set of driver-dedicated system threads, but most FSDs use system worker threads in order to get work done without tying up user-mode threads that make I/O requests. Any FSD might set up its own process address space in which its driver-dedicated threads execute, but the system-supplied FSDs avoid this practice to conserve system memory.

File systems generally use system worker threads to set up and manage internal work queues of IRPs that they send to one or more lower-level drivers, possibly for different devices.

While the lowest-level driver shown in the Processing IRPs in Layered Drivers figure processes each IRP in stages through a set of discrete, driver-supplied routines, it does not use system threads as the file system does. A lowest-level driver does not need its own thread context unless setting up its device for I/O is such a protracted process that it has a noticeable effect on system performance. Few lowest-level or intermediate drivers need to set up their own driver-dedicated or device-dedicated system threads, and those that do pay a performance penalty caused by context switches to their threads.

Most kernel-mode drivers, like the physical device driver in the Processing IRPs in Layered Drivers figure, execute in an arbitrary thread context: that of whatever thread happens to be current when they are called to process an IRP. Consequently, drivers usually maintain state about their I/O operations and the devices they service in a driver-defined part of their device objects, called a *device extension*.

Points to Consider about User I/O Requests

6/25/2019 • 3 minutes to read • Edit Online

Keep the following points in mind when designing a kernel-mode driver:

- Drivers can be layered, and more than one driver can process a single I/O request (IRP).
- A driver cannot make any assumptions about which other drivers will be in the device stack. Therefore, each driver should be prepared to receive requests from any other driver and should handle all potential error cases.
- Drivers communicate the success or failure of a requested I/O operation in the I/O status block of the IRP. The I/O manager communicates the success or failure of a requested I/O operation to a user-mode requester.
- Drivers need not and should not be designed to provide application-specific support. A protected subsystem or its subsystem-specific, user-mode drivers supply this kind of support. There is one exception to this rule: an MS-DOS application that relies on an application-dedicated device can require a kernel-mode driver to control the device and a closely coupled Win32 user-mode virtual device driver (VDD). For more information about VDDs, see the Virtual Device Drivers documentation in the Windows Driver Development Kit (DDK). (The DDK preceded the Windows Driver Kit [WDK].)
- A new driver must handle the same set of **IRP_MJ_*XXX*** as any system-supplied driver it replaces. The I/O manager returns STATUS_INVALID_DEVICE_REQUEST for a given I/O request to a target device if its driver does not define an entry point for that **IRP_MJ_*XXX**. *A device driver also must handle the same I/O control codes for IRP_MJ_DEVICE_CONTROL** requests as any system-supplied driver it replaces. In other words, a new device driver must not "break applications" by implementing less functionality than an existing driver for the same type of device.
- A new intermediate driver inserted into a chain of existing drivers should recognize the same set of **IRP_MJ_*XXX*** as the driver it displaces. The new driver can simply pass on IRPs for those requests that it does not process to the next-lower-level driver. However, a new intermediate driver must not "break the chain" for drivers above and below it by neglecting to define an entry point for an **IRP_MJ_*XXX*** request that the newly displaced, next-lower-level driver does handle.
- A lowest-level driver can access only its own I/O stack location in any IRP that it is sent. A higher-level driver can access only its own and the next-lower-level driver's I/O stack locations in any IRP that it is sent.
- Every driver communicates information to higher-level drivers (and ultimately, to user-mode applications via the I/O manager) only in the I/O status blocks of IRPs because the I/O manager zeros the corresponding I/O stack location as each driver in a chain completes an IRP. Any new driver that attempts to implement back-door communication with a particular higher (or lower) driver compromises its portability and its interoperability with other drivers from one Windows platform or version to the next.
- A pair of drivers can define a set of device-specific (also called *private*) I/O control codes for **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests that the higher of the pair can send down to the lower of the pair. However, such a pair of drivers must follow all of the preceding guidelines if they are to remain portable and interoperable with other drivers from one Windows platform or version to another. If you design a pair of drivers with a private interface, consider the set of I/O control codes to be defined carefully. Make them as generally useful as possible and design your paired drivers to follow the preceding guidelines, so that you (or someone else) can reuse, replace, or displace either or both of your new drivers easily as they migrate from one Windows platform or version to another.

IRP Major Function Codes

7/9/2019 • 2 minutes to read • Edit Online

Each driver-specific I/O stack location (**IO_STACK_LOCATION** for the major function codes that it must support.

The specific operations a driver carries out for a given **IRP_MJ_*XXX*** code depend somewhat on the underlying device, particularly for **IRP_MJ_DEVICE_CONTROL** and **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests. For example, the requests sent to a keyboard driver are necessarily somewhat different from those sent to a disk driver. However, the I/O manager defines the parameters and I/O stack location contents for each system-defined major function code.

Every higher-level driver must set up the appropriate I/O stack location in IRPs for the next-lower-level driver and call **IoCallDriver**, either with each input IRP, or with a driver-created IRP (if the higher-level driver holds on to the input IRP). Consequently, every intermediate driver must supply a dispatch routine for each major function code that the underlying device driver handles. Otherwise, a new intermediate driver will "break the chain" whenever an application or still higher-level driver attempts to send an I/O request down to the underlying device driver.

File system drivers also handle a required subset of system-defined **IRP_MJ_*XXX*** function codes, some with subordinate **IRP_MN_*XXX*** function codes.

Drivers handle IRPs set with some or all of the following major function codes:

IRP_MJ_CLEANUP

IRP_MJ_CLOSE

IRP_MJ_CREATE

IRP_MJ_DEVICE_CONTROL

IRP_MJ_FILE_SYSTEM_CONTROL

IRP_MJ_FLUSH_BUFFERS

IRP_MJ_INTERNAL_DEVICE_CONTROL

IRP_MJ_PNP

IRP_MJ_POWER

IRP_MJ_QUERY_INFORMATION

IRP_MJ_READ

IRP_MJ_SET_INFORMATION

IRP_MJ_SHUTDOWN

IRP_MJ_SYSTEM_CONTROL

IRP_MJ_WRITE

The input and output parameters described in this section are the function-specific parameters in the IRP.

For more information about IRPs, see Handling IRPs.

IRP_MJ_CLEANUP

6/25/2019 • 2 minutes to read • Edit Online

Drivers that maintain process-specific context information must handle cleanup requests in *DispatchCleanup* routines.

When Sent

Receipt of this request indicates that the last handle for a file object that is associated with the target device object has been closed (but, due to outstanding I/O requests, might not have been released).

Input Parameters

None

Output Parameters

None

Operation

This IRP is sent in the context of the process that closed the file object handle. Therefore, the driver should release process-specific resources, such as user memory, that the driver previously locked or mapped.

If the driver's device objects were set up as exclusive, so that only a single thread can use the device at a time, the driver must complete every IRP that is currently queued to the target device object and set STATUS_CANCELLED in each IRP's I/O status block.

Otherwise, the driver must cancel and complete only the currently queued IRPs that are associated with the file object handle that is being released. (A pointer to the file object is located in the **FileObject** member of the driver's **IO_STACK_LOCATION** of the IRP.) After canceling these queued IRPs, the driver completes the cleanup IRP and sets STATUS_SUCCESS in its I/O status block.

For more information about handling this request, see DispatchCleanup Routines.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchCleanup

IO_STACK_LOCATION

IRP_MJ_CLOSE

IRP_MJ_CLOSE

12/5/2018 • 2 minutes to read • Edit Online

Every driver must handle close requests in a *DispatchClose* routine, with the possible exception of a driver whose device cannot be disabled or removed from the machine without bringing down the system. A disk driver whose device holds the system page file is an example of such a driver. Note that the driver of such a device also cannot be unloaded dynamically.

When Sent

Receipt of this request indicates that the last handle of the file object that is associated with the target device object has been closed and released. All outstanding I/O requests have been completed or canceled.

Input Parameters

None

Output Parameters

None

Operation

Many device and intermediate drivers merely set STATUS_SUCCESS in the I/O status block of the IRP and complete the close request. However, what a given driver does on receipt of a close request depends on the driver's design. In general, a driver should undo whatever actions it takes on receipt of the **IRP_MJ_CREATE** request. Device drivers whose device objects are exclusive, such as a serial driver, also can reset the hardware on receipt of a close request.

The **IRP_MJ_CLOSE** request is not necessarily sent in the context of the process that closed the file object handle. If the driver must release process-specific resources, such as user memory, that the driver previously locked or mapped, it must do so in response to an **IRP_MJ_CLEANUP** request.

The **IRP_MJ_CLOSE** request will always be sent at PASSIVE_LEVEL.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchClose

IRP_MJ_CLEANUP

IRP_MJ_CREATE

IRP_MJ_CREATE

6/25/2019 • 2 minutes to read • Edit Online

Every kernel-mode driver must handle **IRP_MJ_CREATE** requests in a *DRIVER_DISPATCH* callback function.

When Sent

The operating system sends an **IRP_MJ_CREATE** request to open a handle to a file object or device object. For example, when a driver calls **ZwCreateFile**, the operating system sends an **IRP_MJ_CREATE** request to perform the actual open operation.

Input Parameters

The **Parameters.Create.SecurityContext** member points to an **IO_SECURITY_CONTEXT** structure that describes the security context for the request. The **Parameters.Create.SecurityContext->DesiredAccess** member is an access mask that specifies the access rights that are being requested by the caller.

The **Parameters.Create.Options** member is a ULONG value that describes the options that are used when opening the handle. The high 8 bits correspond to the value of the *CreateDisposition* parameter of **ZwCreateFile**, and the low 24 bits correspond to the value of the *CreateOptions* parameter of **ZwCreateFile**.

The **Parameters.Create.ShareAccess** member is a USHORT value that describes the type of share access. This value corresponds to the value of the *ShareAccess* parameter of **ZwCreateFile**.

The **Parameters.Create.FileAttributes** and **Parameters.Create.EaLength** members are reserved for use by file systems and file system filter drivers. For more information, see the **IRP_MJ_CREATE** topic in the Installable File System (IFS) documentation.

Output Parameters

None

Operation

Most device and intermediate drivers set STATUS_SUCCESS in the I/O status block of the IRP and complete the create request, but drivers can optionally use their *DRIVER_DISPATCH* callback function to reserve resources for any subsequent I/O requests for that handle. For example, the system serial driver maps its paged-out code and allocates any resources that are necessary to handle subsequent I/O requests for the user-mode thread that is attempting to open the device for input and output.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DRIVER_DISPATCH

DispatchCreateClose

IO_SECURITY_CONTEXT

ZwCreateFile

IRP_MJ_DEVICE_CONTROL

6/25/2019 • 2 minutes to read • Edit Online

Every driver whose device objects belong to a particular device type (see Specifying Device Types) is required to support this request in a *DispatchDeviceControl* routine, if a set of system-defined I/O control codes (IOCTLs) exists for the type. For more info about IOCTLs, see Introduction to I/O Control Codes.

Higher-level drivers usually pass these requests on to an underlying device driver. Each device driver in a driver stack is assumed to support this request, along with a set of device type-specific, public or private IOCTLs. For more information about IOCTLs for specific device types, see device type-specific documentation in the Microsoft Windows Driver Kit (WDK).

When Sent

Any time following the successful completion of a create request.

Input Parameters

The I/O control code is contained at **Parameters.DeviceIoControl.IoControlCode** in the driver's I/O stack location of the IRP.

Other input parameters depend on the I/O control code's value. For more information, see Buffer Descriptions for I/O Control Codes.

Output Parameters

Output parameters depend on the I/O control code's value. For more information, see Buffer Descriptions for I/O Control Codes.

Operation

A driver receives this I/O control code because user-mode thread has called the Microsoft Win32 **DeviceIoControl** function, or a higher-level kernel-mode driver has set up the request. Possibly, a user-mode driver has called **DeviceIoControl**, passing in a driver-defined (also called *private*) I/O control code, to request device- or driver-specific support from a closely coupled, kernel-mode device driver.

On receipt of a device I/O control request, a higher-level driver usually passes the IRP on to the next-lower driver. However, there are some exceptions to this practice. For example, a class driver that has stored configuration information obtained from the underlying port driver might complete certain IOCTL_*XXX* requests without passing the IRP down to the corresponding port driver.

On receipt of a device I/O control request, a device driver examines the I/O control code to determine how to satisfy the request. For most public I/O control codes, device drivers transfer a small amount of data to or from the buffer at **Irp->AssociatedIrp.SystemBuffer**.

For general information about I/O control codes for **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests, see Using I/O Control Codes. See also Device Type-Specific I/O Requests.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchDeviceControl

IRP_MJ_FILE_SYSTEM_CONTROL

6/25/2019 • 2 minutes to read • Edit Online

Only file system drivers process **IRP_MJ_FILE_SYSTEM_CONTROL** requests. For more information about the use of this IRP major function code by file system drivers, see **IRP_MJ_FILE_SYSTEM_CONTROL**. For more information about file system drivers, see File System Drivers.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

IRP_MJ_FLUSH_BUFFERS

12/7/2018 • 2 minutes to read • Edit Online

Drivers of devices with internal caches for data and drivers that maintain internal buffers for data must handle this request in a *DispatchFlushBuffers* routine.

When Sent

Receipt of a flush request indicates that the driver should flush the device's cache or its internal buffer, or, possibly, should discard the data in its internal buffer.

Input Parameters

None

Output Parameters

None

Operation

The driver transfers any data currently cached in the device or held in the driver's internal buffers before completing the flush request. The driver of an input-only device that buffers data internally might simply discard the currently buffered device data before completing the flush IRP, depending on the nature of its device.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchFlushBuffers

IRP_MJ_INTERNAL_DEVICE_CONTROL

6/25/2019 • 2 minutes to read • Edit Online

In general, any replacement for an existing driver that supports internal device control requests should handle this request in a *DispatchInternalDeviceControl* routine. Such a driver must support at least the same set of internal I/O control codes as the driver it replaces. Otherwise, existing higher-level drivers might not work with the new driver.

Drivers that replace certain lower-level system drivers are required to handle this request. For example, a replacement for the system parallel port driver must continue to support existing parallel class drivers. Note that certain system drivers that handle this request cannot be replaced, in particular, the system-supplied SCSI and video port drivers.

When Sent

Any time after the successful completion of a create request.

Input Parameters

The I/O control code is contained at **Parameters.DeviceIoControl.IoControlCode** in the I/O stack location of the IRP.

Other input parameters depend on the I/O control code's value. For more information, see Buffer Descriptions for I/O Control Codes.

Output Parameters

Output parameters depend on the I/O control code's value. For more information, see Buffer Descriptions for I/O Control Codes.

Operation

Drivers receive **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests when another driver calls either **IoBuildDeviceIoControlRequest** or **IoAllocateIrp** to create a request.

This I/O control code has been defined for communication between paired and layered kernel-mode drivers, such as one or more class drivers layered over a port driver. The higher-level driver sets up IRPs with device- or driver-specific I/O control codes, requesting support from the next-lower driver.

The requested operation is device- or driver-specific.

For general information about I/O control codes for **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests, see Using I/O Control Codes. See also Device Type-Specific I/O Requests.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchInternalDeviceControl

IoAllocateIrp

IoBuildDeviceIoControlRequest

IRP_MJ_PNP

12/7/2018 • 2 minutes to read • Edit Online

All drivers must be prepared to service **IRP_MJ_PNP** requests in a *DispatchPnP* routine.

When Sent

The PnP manager sends **IRP_MJ_PNP** requests during enumeration, resource rebalancing, and any other time Plug and Play activity occurs on the system. Drivers can also send certain **IRP_MJ_PNP** requests, depending on the minor function code.

Input Parameters

Depends on the value at **MinorFunction** in the current I/O stack location of the IRP. Every **IRP_MJ_PNP** request specifies a minor function code that identifies the requested PnP action.

Output Parameters

Depends on the value at **MinorFunction** in the current I/O stack location of the IRP.

Operation

See Plug and Play Minor IRPs for detailed information about **IRP_MJ_PNP** requests.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchPnP

IRP_MJ_POWER

12/7/2018 • 2 minutes to read • Edit Online

All drivers must be prepared to service **IRP_MJ_POWER** requests in a *DispatchPower* routine.

When Sent

The power manager or a driver can send **IRP_MJ_POWER** requests at any time the operating system is running.

Input Parameters

Depends on the value at **MinorFunction** in the current I/O stack location of the IRP. Every **IRP_MJ_POWER** request specifies a minor function code that identifies the requested power action.

Output Parameters

Depends on the value at **MinorFunction** in the current I/O stack location of the IRP.

Operation

In addition to the usual rules that govern the processing of IRPs, **IRP_MJ_POWER** IRPs have the following special requirement: A driver that receives a power IRP must not change the major and minor function codes in any I/O stack locations in the IRP that have been set by the power manager or by higher-level drivers. The power manager relies on these function codes remaining unchanged until the IRP is completed. Violations of this rule can cause problems that are difficult to debug. For example, the operating system might stop responding, or "hang."

See Power Management Minor IRPs for detailed information about **IRP_MJ_POWER** requests.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchPower

IRP_MJ_QUERY_INFORMATION

6/25/2019 • 2 minutes to read • Edit Online

Drivers can optionally handle an **IRP_MJ_QUERY_INFORMATION** request.

When Sent

The operating system sends an **IRP_MJ_QUERY_INFORMATION** request to obtain metadata about a file or file handle. For example, when a driver calls **ZwQueryInformationFile**, the operating system sends an **IRP_MJ_QUERY_INFORMATION** request.

Input Parameters

The **Parameters.QueryFile.FileInformationClass** member is a **FILE_INFORMATION_CLASS** constant that specifies the type of metadata to provide. For more information about the types of metadata, see the *FileInformationClass* parameter of the **ZwQueryInformationFile** routine.

The **Parameters.QueryFile.Length** member specifies the length of the buffer that the **AssociatedIrp.SystemBuffer** member points to.

Output Parameters

The **AssociatedIrp.SystemBuffer** member points to the buffer where the driver supplies the requested information. The value of **Parameters.QueryFile.FileInformationClass** determines the format of the metadata (a **FILE_*XXX*_INFORMATION** structure) to return. For more information about the formats of metadata, see the **FILE_INFORMATION_CLASS** enumeration.

Operation

Drivers are not required to handle this request, and drivers that do are not required to handle every possible value of **Parameters.QueryFile.FileInformationClass**. The driver's dispatch routine should return an error code such as STATUS_INVALID_DEVICE_REQUEST for any values that it does not handle.

Not all of the possible values of **FILE_INFORMATION_CLASS** can occur.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

ZwQueryInformationFile

IRP_MJ_READ

6/25/2019 • 2 minutes to read • Edit Online

Every device driver that transfers data from its device to the system must handle read requests in a *DispatchRead* or *DispatchReadWrite* routine, as must any higher-level driver layered over such a device driver.

When Sent

Any time following the successful completion of a create request.

Possibly, a user-mode application or Win32 component with a handle for the file object representing the target device object has requested a data transfer from the device. Possibly, a higher-level driver has created and set up the read IRP.

Input Parameters

The driver's I/O stack location in the IRP indicates how many bytes to transfer at **Parameters.Read.Length**.

Some drivers use the value at **Parameters.Read.Key** to sort incoming read requests into a driver-determined order in the device queue or in a driver-managed internal queue of IRPs.

Certain types of drivers also use the value at **Parameters.Read.ByteOffset**, which indicates the starting offset for the transfer operation. For example, see the **IRP_MJ_READ** topic in the Installable File System (IFS) documentation.

Output Parameters

Depending on whether the underlying device driver sets up the target device object's **Flags** with DO_BUFFERED_IO or with DO_DIRECT_IO, data is transferred into one of the following:

- The buffer at **Irp->AssociatedIrp.SystemBuffer** if the driver uses buffered I/O.
- The buffer described by the MDL at **Irp->MdlAddress** if the underlying device driver uses direct I/O (DMA or PIO).

Operation

On receipt of a read request, a higher-level driver sets up the I/O stack location in the IRP for the next-lower driver, or it creates and sets up additional IRPs for one or more lower drivers. It can set up its *IoCompletion* routine, which is optional for the input IRP but required for driver-created IRPs, by calling **IoSetCompletionRoutine**. Then, the driver passes the request on to the next-lower driver with **IoCallDriver**.

On receipt of a read request, a device driver transfers data from its device to system memory. The device driver sets the **Information** field of the I/O status block to the number of bytes transferred when it completes the IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchRead

DispatchReadWrite

IoCallDriver

IoSetCompletionRoutine

IRP_MJ_SET_INFORMATION

6/25/2019 • 2 minutes to read • Edit Online

Device drivers can optionally handle an **IRP_MJ_SET_INFORMATION** request.

When Sent

The operating system sends an **IRP_MJ_SET_INFORMATION** request to set metadata about a file or file handle. For example, when a driver calls **ZwSetInformationFile**, the operating system sends an **IRP_MJ_SET_INFORMATION** request.

Input Parameters

The **Parameters.SetFile.FileInformationClass** member is a **FILE_INFORMATION_CLASS** constant that specifies the type of metadata to set. For more information about the types of metadata, see the *FileInformationClass* parameter of **ZwSetInformationFile**.

The **Parameters.SetFile.Length** member specifies the length of the buffer that the **AssociatedIrp.SystemBuffer** member points to.

AssociatedIrp.SystemBuffer points to the buffer that contains the new information setting. The value of **Parameters.SetFile.FileInformationClass** determines the format of the data (a **FILE_*XXX*_INFORMATION** structure) to return. For more information about the formats of metadata, see the **FILE_INFORMATION_CLASS** enumeration.

Output Parameters

None

Operation

Drivers are not required to handle this request, and drivers that do are not required to handle every possible value of **Parameters.SetFile.FileInformationClass**. The driver's dispatch routine should return an error code such as STATUS_INVALID_DEVICE_REQUEST for any values that it does not handle.

Not all of the possible values of **FILE_INFORMATION_CLASS** can occur.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

ZwSetInformationFile

IRP_MJ_SHUTDOWN

10/15/2019 • 2 minutes to read • Edit Online

Drivers of mass-storage devices that have internal caches for data must handle this request in a *DispatchShutdown* routine. Drivers of mass-storage devices and intermediate drivers layered over them also must handle this request if an underlying driver maintains internal buffers for data.

When Sent

Receipt of a shutdown request indicates that a file system driver is sending notice that the system is being shut down.

One or more file system drivers can send such a lower-level driver more than one shutdown request when a user logs off or when the system is being shut down for some other reason.

The PnP manager sends this IRP at IRQL<=APC_LEVEL in an arbitrary thread context.

Input Parameters

None

Output Parameters

None

Operation

The driver must complete the transfer of any data currently cached in the device or held in the driver's internal buffers before completing the shutdown request.

A driver does not receive an **IRP_MJ_SHUTDOWN** request for a device object unless it registers to do so with either **IoRegisterShutdownNotification** or **IoRegisterLastChanceShutdownNotification**.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchShutdown

IoRegisterLastChanceShutdownNotification

IoRegisterShutdownNotification

IRP_MJ_SYSTEM_CONTROL

6/25/2019 • 2 minutes to read • Edit Online

All drivers must provide a *DispatchSystemControl* routine that handles **IRP_MJ_SYSTEM_CONTROL** requests, which are sent by the kernel-mode component of Windows Management Instrumentation (WMI).

When Sent

The WMI kernel-mode component can send an **IRP_MJ_SYSTEM_CONTROL** request any time following a driver's successful registration as a supplier of WMI data. WMI IRPs typically are sent when a user-mode data consumer has requested WMI data.

Input Parameters

Depends on the value at **MinorFunction** in the current I/O stack location of the IRP. Every **IRP_MJ_SYSTEM_CONTROL** request specifies a minor function code that identifies the requested WMI action.

Output Parameters

Depends on the value at **MinorFunction** in the current I/O stack location of the IRP.

Operation

All drivers must support **IRP_MJ_SYSTEM_CONTROL** requests by supplying a *DispatchSystemControl* routine.

Drivers that support Windows Management Instrumentation (WMI) must handle **IRP_MJ_SYSTEM_CONTROL** requests by processing the minor function codes associated with this major function code. For information about the WMI minor function codes, see WMI Minor IRPs.

Drivers that do not support WMI by registering as a WMI data provider must pass **IRP_MJ_SYSTEM_CONTROL** requests to the next lower driver.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchSystemControl

IRP_MJ_WRITE

6/25/2019 • 2 minutes to read • Edit Online

Every device driver that transfers data from the system to its device must handle write requests in a *DispatchWrite* or *DispatchReadWrite* routine, as must any higher-level driver layered over such a device driver.

When Sent

Any time following the successful completion of a create request.

Possibly, a user-mode application or Win32 component with a handle for the file object representing the target device object has requested a data transfer to the device. Possibly, a higher-level driver has created and set up the write IRP.

Input Parameters

The driver's I/O stack location in the IRP indicates how many bytes to transfer at **Parameters.Write.Length**.

Some drivers use the value at **Parameters.Write.Key** to sort incoming write requests into a driver-determined order in the device queue or in a driver-managed internal queue of IRPs.

Certain types of drivers also use the value at **Parameters.Write.ByteOffset**, which indicates the starting offset for the transfer operation. For example, see the **IRP_MJ_WRITE** topic in the Installable File System (IFS) documentation.

Depending on whether the underlying device driver sets up the target device object's **Flags** with DO_BUFFERED_IO or with DO_DIRECT_IO, data is transferred from one of the following:

- The buffer at **Irp->AssociatedIrp.SystemBuffer**, if the driver uses buffered I/O
- The buffer described by the MDL at **Irp->MdlAddress**, if the underlying device driver uses direct I/O (DMA or PIO)

Output Parameters

None

Operation

On receipt of a write request, a higher-level driver sets up the I/O stack location in the IRP for the next-lower driver, or it creates and sets up additional IRPs for one or more lower drivers. It can set up its *IoCompletion* routine, which is optional for the input IRP but required for driver-created IRPs, by calling **IoSetCompletionRoutine**. Then, the driver passes the request on to the next-lower driver with **IoCallDriver**.

On receipt of a write request, a device driver transfers data from system memory to its device. The device driver sets the **Information** field of the I/O status block to the number of bytes transferred when it completes the IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchReadWrite

DispatchWrite

IoCallDriver

IoCompletion

IoSetCompletionRoutine

I/O Stack Locations

6/25/2019 • 4 minutes to read • Edit Online

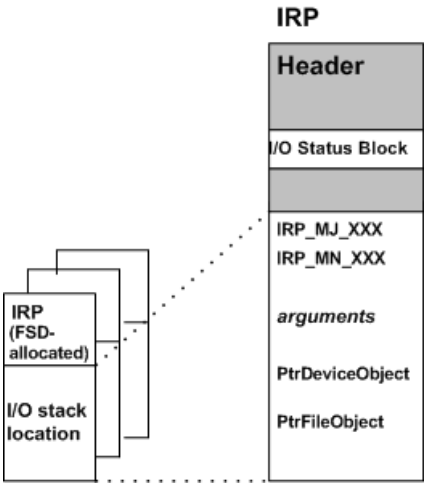
The I/O manager gives each driver in a chain of layered drivers an I/O stack location for every IRP that it sets up. Each I/O stack location consists of an **IO_STACK_LOCATION** structure.

The I/O manager creates an array of I/O stack locations for each IRP, with an array element corresponding to each driver in a chain of layered drivers. Each driver owns one of the stack locations in the packet and calls **IoGetCurrentIrpStackLocation** to obtain driver-specific information about the I/O operation.

Each driver in such a chain is responsible for calling **IoGetNextIrpStackLocation**, then setting up the next-lower driver's I/O stack location. Any higher-level driver's I/O stack location can also be used to store context about an operation so that the driver's *IoCompletion* routine can perform its cleanup operations.

The Processing IRPs in Layered Drivers figure shows two I/O stack locations in the original IRP because it shows two drivers, a file system driver and a mass-storage device driver. The driver-allocated IRPs in the Processing IRPs in Layered Drivers figure do not have a stack location for the FSD that created them. Any higher-level driver that allocates IRPs for lower-level drivers also determines how many I/O stack locations the new IRPs should have, according to the **StackSize** value of the next-lower driver's device object.

The following figure shows the contents of the IRP in more detail.



As shown in the figure, each driver-specific I/O stack location in an IRP contains the following general information:

- The major function code (**IRP_MJ_*XXX***), indicating the basic operation the driver should carry out
- For some major function codes handled by FSDs, higher-level SCSI drivers, and all PnP drivers, a minor function code (**IRP_MN_*XXX***), indicating which subcase of the basic operation the driver should carry out
- A set of operation-specific arguments, such as the length and starting location of a buffer into which or from which the driver transfers data
- A pointer to the driver-created device object, representing the target (physical, logical, or virtual) device for the requested operation
- A pointer to the file object, representing an open file, device, directory, or volume

  A file system driver accesses the file object through its I/O stack location in IRPs. Other drivers usually ignore the file object.

The set of IRP major and minor function codes that a particular driver handles can be device-type-specific. However, lowest-level drivers and intermediate drivers (including PnP function and filter drivers) usually handle the following set of basic requests:

- **IRP_MJ_CREATE** — open the target device object, indicating that it is present and available for I/O operations
- **IRP_MJ_READ** — transfer data from the device
- **IRP_MJ_WRITE** — transfer data to the device
- **IRP_MJ_DEVICE_CONTROL** — set up (or reset) the device, according to a system-defined, device-type-specific I/O control code (IOCTL)
- **IRP_MJ_CLOSE** — close the target device object
- **IRP_MJ_PNP** — perform a Plug and Play operation on the device. An **IRP_MJ_PNP** request is sent by the PnP manager through the I/O manager.
- **IRP_MJ_POWER** — perform a power operation on the device. An **IRP_MJ_POWER** request is sent by the power manager through the I/O manager.

For more information about the major IRP function codes that drivers are required to handle, see IRP Major Function Codes.

In general, the I/O manager sends IRPs with at least two I/O stack locations to mass-storage device drivers because a file system is layered over other drivers for mass-storage devices. The I/O manager sends IRPs with a single stack location to any driver that has no other driver layered above it.

However, the I/O manager provides support for adding a new driver to any chain of existing drivers in the system. For example, an intermediate *mirror driver* that backs up data on a given disk partition might be inserted between a pair of drivers, such as the file system driver and lowest-level driver shown in the Processing IRPs in Layered Drivers figure. When this new driver attaches itself to the device stack, the I/O manager adjusts the number of I/O stack locations in all IRPs it sends to the file system, mirror, and lowest-level drivers. Every IRP that the file system in the Processing IRPs in Layered Drivers figure allocated would also contain another I/O stack location for such a new mirror driver.

Note that this support for adding new drivers to an existing chain implies certain restrictions on any particular driver's access to the I/O stack locations in IRPs:

- A higher-level driver in a chain of layered drivers can safely access only its own and the next-lower-level driver's I/O stack locations in any IRP. Such a driver must set up the I/O stack location for the next-lower-level driver in IRPs. However, when designing such a higher-level driver, you cannot predict when (or whether) a new driver will be added to the existing chain just below your driver.

  Therefore, you should assume that any subsequently added driver will handle the same IRP major function codes (**IRP_MJ_*XXX***) as the displaced next-lower-level driver did.

- The lowest-level driver in a chain of layered drivers can safely access only its own I/O stack location in any IRP. When designing such a driver, you cannot predict when (or whether) a new driver will be added to the existing chain above your device driver.

  In designing a lowest-level driver, assume that the driver can continue to process IRPs using the information passed in its own I/O stack location, whatever the originating source of a given IRP and however many drivers are layered above it.

I/O Status Blocks

6/25/2019 • 2 minutes to read • Edit Online

An I/O status block, which consists of an **IO_STATUS_BLOCK** structure, is a part of each **IRP**. An I/O status block serves two purposes:

- It provides a higher-level driver's *IoCompletion* routine a way of determining whether the service worked when the IRP is completed.
- It provides more information about why the service either worked or did not work.

Upon completion of a IRP, the **Status** field indicates whether the drivers that processed the IRP actually satisfied the request or failed the IRP with an error status. The **Information** field supplies the caller with more information about what actually occurred. For example, it contains the number of bytes actually transferred after a read or write operation.

For more information, see Setting the I/O Status Block in an IRP.

Passing IRPs down the Driver Stack

6/25/2019 • 2 minutes to read • Edit Online

When a driver's dispatch routine receives an IRP, it must call **IoGetCurrentIrpStackLocation** so that it can check its own I/O stack location and determine that any parameters are valid. If the driver cannot satisfy and complete the request itself, it can do one of the following:

- Pass the IRP on for further processing by lower-level drivers.
- Create one or more new IRPs and pass them down to lower-level drivers.

A higher-level driver should pass an I/O request on to a next-lower driver as follows:

1. If the driver will pass the input IRP on to the next lower-level driver, the dispatch routine should call **IoSkipCurrentIrpStackLocation** or **IoCopyCurrentIrpStackLocationToNext** to set up the I/O stack location of the next-lower driver.

   If the driver calls **IoAllocateIrp** to allocate one or more additional IRPs for lower drivers, the dispatch routine must initialize the next-lower driver's I/O stack location by following the steps that are described in Processing IRPs in an Intermediate-Level Driver.

   The dispatch routine can modify some of the parameters in the next-lower driver's I/O stack location for certain requests. For example, a higher-level driver might modify the parameters for a large transfer request when the underlying device has a known limit in transfer capacity, and reuse the IRP to send partial-transfer requests to the underlying device driver.

2. Call **IoSetCompletionRoutine**.

   If the dispatch routine is passing a received IRP to the next-lower driver, setting an *IoCompletion* routine is optional but useful, because the routine can perform such tasks as determining how lower drivers completed the request, reusing the IRP for partial transfers, updating whatever state the driver maintains if it tracks IRPs, and retrying a request that was returned with an error.

   If the dispatch routine has allocated new IRPs, setting an *IoCompletion* routine is required because the routine must release each IRP after lower drivers have completed it.

   For more information about *IoCompletion* routines, see Completing IRPs.

3. Call **IoCallDriver** with each IRP to be processed by lower drivers.

4. Return an appropriate NTSTATUS value, such as:

   - STATUS_PENDING

     The driver usually returns STATUS_PENDING if the input IRP is an asynchronous request, such as **IRP_MJ_READ** or **IRP_MJ_WRITE**.

   - The result of the call to **IoCallDriver**

     The driver frequently returns the result of the call to **IoCallDriver** if the input IRP is a synchronous request, such as **IRP_MJ_CREATE**.

A lowest-level device driver passes any IRP that it cannot complete in its dispatch routine on to other driver routines as follows:

1. Call **IoMarkIrpPending** with the input IRP.

2. Call **IoStartPacket** to pass on or queue the IRP to the driver's *StartIo* routine, unless the driver manages

its own internal IRP queuing, as described in Driver-Managed IRP Queues.

If the driver does not have a *StartIo* routine but handles cancelable IRPs, it must either register a *Cancel* routine or implement a cancel-safe IRP queue. For more information about *Cancel* routines, see Canceling IRPs.

3. Return STATUS_PENDING.

Creating IRPs for Lower-Level Drivers

6/25/2019 • 4 minutes to read • Edit Online

To allocate an IRP for an asynchronous request, which will be processed in an arbitrary thread context by lower drivers, a *DispatchReadWrite* routine can call one of the following support routines:

- **IoAllocateIrp**, which allocates an IRP and a number of zero-initialized I/O stack locations

  The dispatch routine must set up the next-lower driver's I/O stack location for the newly allocated IRP, usually by copying (possibly modified) information from its own stack location in the original IRP. If a higher-level driver allocates an I/O stack location of its own for a newly-allocated IRP, the dispatch routine can set up per-request context information there for the *IoCompletion* routine to use.

- **IoBuildAsynchronousFsdRequest**, which sets up the next-lower driver's I/O stack location for the caller, according to caller-specified parameters

  Higher-level drivers can call this routine to allocate IRPs for **IRP_MJ_READ**, **IRP_MJ_WRITE**, **IRP_MJ_FLUSH_BUFFERS**, and **IRP_MJ_SHUTDOWN** requests.

  When an *IoCompletion* routine is called for such an IRP, it can check the I/O status block, and if necessary (or possible) set up the next-lower driver's I/O stack location in the IRP again and retry the request or reuse it. However, the *IoCompletion* routine has no local context storage for itself in the IRP, so the driver must maintain context about the original request elsewhere in resident memory.

- **IoMakeAssociatedIrp**, which allocates an IRP and a number of zero-initialized I/O stack locations, and associates the IRP with a *master* IRP.

  Intermediate drivers cannot call **IoMakeAssociatedIrp** to create IRPs for lower drivers.

  Any highest-level driver that calls **IoMakeAssociatedIrp** to create IRPs for lower drivers can return control to the I/O manager after sending its associated IRPs on and calling **IoMarkIrpPending** for the original, master IRP. A highest-level driver can rely on the I/O manager to complete the master IRP when all associated IRPs have been completed by lower drivers.

  Drivers seldom set an *IoCompletion* routine for an associated IRP. If a highest-level driver calls **IoSetCompletionRoutine** for an associated IRP it creates, the I/O manager does not complete the master IRP if the driver returns STATUS_MORE_PROCESSING_REQUIRED from its *IoCompletion* routine. In these circumstances, the driver's *IoCompletion* routine must explicitly complete the master IRP with **IoCompleteRequest**.

If a driver allocates an I/O stack location of its own in a new IRP, the dispatch routine must call **IoSetNextIrpStackLocation** before it calls **IoGetCurrentIrpStackLocation** to set up context in its own I/O stack location for the *IoCompletion* routine. For more information, see Processing IRPs in an Intermediate-Level Driver.

The dispatch routine must call **IoMarkIrpPending** with the original IRP, but not with any driver-allocated IRPs because the *IoCompletion* routine will free them.

If the dispatch routine is allocating IRPs for partial transfers and the underlying device driver might control a removable-media device, the dispatch routine must set up the thread context in its newly allocated IRPs from the value at **Tail.Overlay.Thread** in the original IRP.

An underlying driver for a removable-media device might call **IoSetHardErrorOrVerifyDevice**, which references the pointer at **Irp->Tail.Overlay.Thread**, for a driver-allocated IRP. If the driver calls this support routine, the file

system driver can send a dialog box to the appropriate user thread that prompts the user to cancel, retry, or fail an operation that the driver could not satisfy. See Supporting Removable Media for more information.

Dispatch routines must return STATUS_PENDING after sending all driver-allocated IRPs on to lower drivers.

A driver's *IoCompletion* routine should free all driver-allocated IRPs with **IoFreeIrp** before it calls **IoCompleteRequest** for the original IRP. When it completes the original IRP, the *IoCompletion* routine must free all driver-allocated IRPs before it returns control.

Each higher-level driver sets up any driver-allocated (and reused) IRPs for lower drivers in such a way that it is immaterial to the underlying device driver whether a given request comes from an intermediate driver or originates from any other source, such as a file system or user-mode application.

Highest-level drivers can call **IoMakeAssociatedIrp** to allocate IRPs and set them up for a chain of lower drivers. The I/O manager automatically completes the original IRP when all its associated IRPs have been completed, as long as the driver does not call **IoSetCompletionRoutine** with the original IRP or with any of the associated IRPs it allocates. Highest-level drivers must not, however, allocate associated IRPs for any IRP that requests a buffered I/O operation.

An intermediate-level driver cannot allocate IRPs for lower-level drivers by calling **IoMakeAssociatedIrp**. Any IRP an intermediate driver receives might already be an associated IRP, and a driver cannot associate another IRP with such an IRP.

Instead, if an intermediate driver creates IRPs for lower drivers, it should call **IoAllocateIrp**, **IoBuildDeviceIoControlRequest**, **IoBuildSynchronousFsdRequest**, or **IoBuildAsynchronousFsdRequest**. However, **IoBuildSynchronousFsdRequest** can be called only in the following circumstances:

- By a driver-created thread to build IRPs for read or write requests, because such a thread can wait in a nonarbitrary thread context (its own) on a dispatcher object, such as a driver-initialized *Event* passed to **IoBuildSynchronousFsdRequest**
- In the system thread context during initialization or while unloading
- To build IRPs for inherently synchronous operations, such as create, flush, shutdown, close, and device control requests

However, a driver is more likely to call **IoBuildDeviceIoControlRequest** to allocate device control IRPs than **IoBuildSynchronousFsdRequest**.

Queuing and Dequeuing IRPs

6/25/2019 • 3 minutes to read • Edit Online

Because the I/O manager supports asynchronous I/O within a multitasking and multithreaded system, I/O requests to a device can come in faster than its driver can process them to completion, particularly in multiprocessor machines. Consequently, IRPs bound to any particular device must be queued in the driver when its device is already busy processing another IRP.

Therefore, a lowest-level driver requires one of the following:

- A *StartIo* routine, which the I/O manager calls to start I/O operations for IRPs the driver has queued to a system-supplied IRP queue (see **IoStartPacket**).
- An internal IRP queuing and dequeuing mechanism, which the driver uses to manage IRPs that come in faster than it can satisfy them. Drivers can use device queues, interlocked queues, or cancel-safe queues. For more information, see Driver-Managed IRP Queues.

Only a lowest-level device driver that can satisfy and complete every possible IRP in its dispatch routines needs no *StartIo* routine and no driver-managed queues for IRPs.

Higher-level drivers almost never have *StartIo* routines. Most intermediate drivers have neither *StartIo* routines nor internal queues; an intermediate driver can usually pass IRPs with valid parameters on from its dispatch routines and do whatever postprocessing is required for any IRP in its *IoCompletion* routine.

The following describes, in general, some of the design considerations for determining whether to implement a *StartIo* routine with or without internal, driver-managed queues for IRPs.

StartIo Routines in Drivers

For computer peripheral devices capable of handling only one device I/O operation at a time, device drivers can implement *StartIo* routines. For these drivers, the I/O manager provides **IoStartPacket** and **IoStartNextPacket** routines to queue and dequeue IRPs to and from a system-supplied IRP queue.

For more information about *StartIo* routines, see Writing a StartIo Routine.

Internal Queues for IRPs in Drivers

If a device can support more than one concurrent I/O operation, its lowest-level device driver must set up internal request queues and manage its own queuing of IRPs. For example, the system serial driver maintains separate queues for read, write, purge, and wait operations on its devices because it supports full-duplex serial devices.

A higher-level driver that sends requests to some number of underlying device drivers also might maintain internal queues of IRPs. For example, file system drivers almost always have internal queues for IRPs.

For more information, see Driver-Managed IRP Queues.

Internal Queue Synchronization

Drivers with device-dedicated threads and highest-level drivers that use executive worker threads (including most file system drivers) usually set up their own queue for IRPs. The queue is shared by the driver thread or driver-supplied worker-thread callback and by other driver routines that process IRPs.

A driver that implements its own queue structure must ensure that access to the queue is synchronized, and that canceled IRPs are removed from the queue. To make this task simpler for driver writers, cancel-safe IRP queues provide a standard framework you can use when implementing an IRP queue. See Cancel-Safe IRP Queues for more information. This is the preferred method for implementing an IRP queue.

Drivers can also implement all IRP queue synchronization and cancel logic explicitly. For example, a driver could use an interlocked queue. The driver's dispatch routines insert IRPs into the interlocked queue and a driver-created thread or the driver's worker-thread callback removes them by calling the **ExInterlocked*Xxx*List** support routines.

For example, the system floppy controller driver uses an interlocked queue. Its device-dedicated thread handles the same processing of IRPs that is done by other device drivers' *StartIo* routines and some of the same processing of IRPs that is done by other device drivers' *DpcForIsr* routines.

Internal Queues with StartIo Routines in Drivers

A driver that manages its own internal queues can also have a *StartIo* routine, but need not. Most lowest-level device drivers either have a *StartIo* routine or manage their own queuing of IRPs, but not both.

An exception to this is the SCSI port driver, which has a *StartIo* routine and manages internal queues of IRPs. The I/O manager queues IRPs to the port driver's *StartIo* routine in the device queue associated with the driver-created device object that represents a SCSI HBA. The SCSI port driver also sets up and manages device queues for IRPs to each target device (corresponding to a SCSI logical unit) on any HBA-driven SCSI bus in the machine.

The SCSI port driver uses its supplemental device queues to hold IRPs sent down from the SCSI class drivers in LU-specific queues whenever any device on a SCSI bus is particularly busy. In effect, this driver's supplemental, LU-specific device queues allow the SCSI port driver to serialize operations for heterogeneous SCSI devices through an HBA while keeping each device on that HBA's SCSI buses as busy as possible.

Writing a StartIo Routine

6/25/2019 • 2 minutes to read • Edit Online

As its name suggests, a *StartIo* routine is responsible for starting an I/O operation on the physical device.

Most lowest-level drivers provide a *StartIo* routine and rely on the I/O manager to queue IRPs to a system-supplied device queue. Some lowest-level drivers are designed to set up and manage their own supplemental IRP queues, but even these usually also provide a *StartIo* routine. (For more information about supplemental queues, see Setting up and Using Device Queues and Managing Device Queues.)

Higher-level drivers, including FSDs and PnP function and filter drivers, seldom have a *StartIo* routine because it can hamper performance. Instead, most file system drivers set up and maintain internal queues of IRPs. Other higher-level drivers either have internal queues for IRPs or simply pass IRPs on to lower drivers from their dispatch routines. See Driver-Managed IRP Queues for more information.

You can use the **IoSetStartIoAttributes** routine to set attributes that modify *StartIo* handling for your driver.

This section contains the following topics:

StartIo Routines in Lowest-Level Drivers

StartIo Routines in Higher-Level Drivers

Points to Consider for StartIo Routines

StartIo Routines in Lowest-Level Drivers

6/25/2019 • 10 minutes to read • Edit Online

The I/O manager's call to a driver's dispatch routine is the first stage in satisfying a device I/O request. The *StartIo* routine is the second stage. Every device driver with a *StartIo* routine is likely to call **IoStartPacket** from its *DispatchRead* and *DispatchWrite* routines, and usually for a subset of the I/O control codes it supports in its *DispatchDeviceControl* routine. The **IoStartPacket** routine adds the IRP to the device's system-supplied device queue or, if the queue is empty, immediately calls the driver's *StartIo* routine to process the IRP.

You can assume that when a driver's *StartIo* routine is called, the target device is not busy. This is because the I/O manager calls *StartIo* under two circumstances; either one of the driver's dispatch routines has just called **IoStartPacket** and the device queue was empty, or the driver's *DpcForIsr* routine is completing another request and has just called **IoStartNextPacket** to dequeue the next IRP.

Before the *StartIo* routine in a highest-level device driver is called, that driver's dispatch routine should have probed and locked down the user buffer, if necessary, to set up valid mapped buffer addresses in the IRP queued to its *StartIo* routine. If a highest-level device driver sets up its device objects for direct I/O (or for neither buffered nor direct I/O), the driver cannot defer locking down a user buffer to its *StartIo* routine; every *StartIo* routine is called in an arbitrary thread context at IRQL = DISPATCH_LEVEL.

Note Any buffer memory to be accessed by a driver's *StartIo* routine must be locked down or allocated from resident, system-space memory and must be accessible in an arbitrary thread context.

In general, any lower-level device driver's *StartIo* routine is responsible for calling **IoGetCurrentIrpStackLocation** with the input IRP and then doing whatever request-specific processing is necessary to start the I/O operation on its device. Request-specific processing can include the following:

- Setting up or updating any state information about the current request that the driver maintains. The state information might be stored in the device extension of the target device object or elsewhere in nonpaged pool allocated by the driver.

  For example, if a device driver maintains an InterruptExpected Boolean for the current transfer operation, its *StartIo* routine might set this variable to **TRUE**. If the driver maintains a time-out counter for the current operation, its *StartIo* routine might set up this value, or the *StartIo* routine might queue the driver's *CustomTimerDpc* routine.

  If the *StartIo* routine shares access to state information or hardware resources with other driver routines, the state information or resource must be protected by a spin lock. (See Spin Locks.)

  If the *StartIo* routine shares access to state information or resources with the driver's *InterruptService* routine, *StartIo* must use **KeSynchronizeExecution** to call a *SynchCritSection* routine that accesses the state or resource information. (See Using Critical Sections.)

- Assigning a sequence number to the IRP in case the driver must log a device I/O error while processing the IRP.

  See Logging Errors for more information.

- If necessary, translating the parameters in the driver's I/O stack location into device-specific values.

  For example, a disk driver might need to calculate the starting sector or byte offset to the physical disk address for a transfer operation, and whether the requested length of the transfer will cross a particular sector boundary or exceed the transfer capacity of its physical device.

- If the driver controls a removable-media device, checking for media changes before programming the device for I/O and notifying its overlying file system if the media has changed.

  For more information, see Supporting Removable Media.

- If the device uses DMA, checking whether the requested **Length** (number of bytes to be transferred, found in the driver's I/O stack location of the IRP) should be split into partial-transfer operations, as explained in Input/Output Techniques, assuming a closely coupled higher-level driver does not presplit large transfers for the device driver.

  The *StartIo* routine of such a device driver also can be responsible for calling **KeFlushIoBuffers** and, if the driver uses packet-based DMA, for calling **AllocateAdapterChannel** with the driver's *AdapterControl* routine.

  See Adapter Objects and DMA, and Maintaining Cache Coherency, for additional details.

- If the device uses PIO, mapping the base virtual address of the buffer, described in the IRP at **Irp->MdlAddress**, to a system-space address with **MmGetSystemAddressForMdlSafe**.

  For read requests, the device driver's *StartIo* routine can be responsible for calling **KeFlushIoBuffers** before PIO operations begin. See Maintaining Cache Coherency for more information.

- If a non-WDM driver uses a controller object, calling **IoAllocateController** to register its *ControllerControl* routine.

- If the driver handles cancelable IRPs, checking whether the input IRP has already been canceled.

- If an input IRP can be canceled before it is processed to completion, the *StartIo* routine must call **IoSetCancelRoutine** with the IRP and the entry point of the driver's *Cancel* routine. The *StartIo* routine must acquire the cancel spin lock for its call to **IoSetCancelRoutine**. Alternatively, a driver can use **IoSetStartIoAttributes** to set the *NonCancelable* attribute for the *StartIo* routine to **TRUE**. This prevents the system from trying to cancel an IRP that has been passed to *StartIo* by a call to **IoStartPacket**.

As a general rule, a driver that uses buffered I/O has a simpler *StartIo* routine than one that uses direct I/O. Drivers that use buffered I/O transfer small amounts of data for each transfer request, while those that use direct I/O (whether DMA or PIO) transfer large amounts of data to or from locked-down buffers that can span physical page boundaries in system memory.

Higher-level drivers layered above physical device drivers usually set up their device objects to match those of their respective device drivers. However, a highest-level driver, particularly a file system driver, can set up device objects for neither direct nor buffered I/O.

Drivers that set up their device objects for buffered I/O can rely on the I/O manager to pass valid buffers in all IRPs it sends to the driver. Lower-level drivers that set up device objects for direct I/O can rely on the highest-level driver in their chain to pass valid buffers in all IRPs sent through any intermediate drivers to the underlying lower-level device driver.

Using Buffered I/O in StartIo Routines

If a driver's *DispatchRead*, *DispatchWrite*, or *DispatchDeviceControl* routine determines that a request is valid and calls **IoStartPacket**, the I/O manager calls the driver's *StartIo* routine to process the IRP immediately if the device queue is empty. If the queue is not empty, **IoStartPacket** queues the IRP. Eventually, a call to **IoStartNextPacket** from the driver's *DpcForIsr* or *CustomDpc* routine causes the I/O manager to dequeue the IRP and call the driver's *StartIo* routine.

The *StartIo* routine calls **IoGetCurrentIrpStackLocation** and determines which operation must be performed to satisfy the request. It preprocesses the IRP in any way necessary before programming the physical device to carry out the I/O request.

If access to the physical device (or the device extension) must be synchronized with an *InterruptService* routine, the *StartIo* routine must call a *SynchCritSection* routine to perform the necessary device programming. For more information, see Using Critical Sections.

A physical device driver that uses buffered I/O transfers data either to or from a system-space buffer, allocated by the I/O manager, that the driver finds in each IRP at **Irp->AssociatedIrp.SystemBuffer**.

Using Direct I/O in StartIo Routines

If a driver's *DispatchRead*, *DispatchWrite*, or *DispatchDeviceControl* routine determines that a request is valid and calls **IoStartPacket**, the I/O manager calls the driver's *StartIo* routine to process the IRP immediately if the device queue is empty. If the queue is not empty, **IoStartPacket** queues the IRP. Eventually, a call to **IoStartNextPacket** from the driver's *DpcForIsr* or *CustomDpc* routine causes the I/O manager to dequeue the IRP and call the driver's *StartIo* routine.

The *StartIo* routine calls **IoGetCurrentIrpStackLocation** and determines which operation must be performed to satisfy the request. It preprocesses the IRP in any way necessary, such as splitting up a large DMA transfer request into partial-transfer ranges and saving state about the **Length** of an incoming transfer request that must be split. Then it programs the physical device to carry out the I/O request.

If access to the physical device (or the device extension) must be synchronized with the driver's ISR, the *StartIo* routine must use a driver-supplied *SynchCritSection* routine to perform the necessary programming. For more information, see Using Critical Sections.

Any driver that uses direct I/O either reads data into or writes data from a locked-down buffer, described by a memory descriptor list (MDL), that the driver finds in the IRP at **Irp->MdlAddress**. Such a driver commonly uses buffered I/O for device control requests. For more information, see Handling I/O Control Requests in StartIo Routines.

The MDL type is an opaque type that drivers do not access directly. Instead, drivers that use PIO remap user-space buffers by calling **MmGetSystemAddressForMdlSafe** with **Irp->MdlAddress** as a parameter. Drivers that use DMA also pass **Irp->MdlAddress** to support routines during their transfer operations to have the buffer addresses remapped to logical ranges for their devices.

Unless a closely coupled higher-level driver splits up large DMA transfer requests for the underlying device driver, a lowest-level device driver's *StartIo* routine must split up each transfer request that is larger than its device can manage in a single transfer operation. Drivers that use system DMA are required to split transfer requests that are too large for the system DMA controller or for their devices to handle in a single transfer operation.

If the device is a subordinate DMA device, its driver must synchronize transfers through a system DMA controller with a driver-allocated adapter object, representing the DMA channel, and a driver-supplied *AdapterControl* routine. The driver of a bus-master DMA device also must use a driver-allocated adapter object to synchronize its transfers and must supply an *AdapterControl* routine if it uses the system's packet-based DMA support, or an *AdapterListControl* routine if it uses the system's scatter/gather support.

Depending on the driver's design, it might synchronize transfer and device control operations on a physical device with a controller object and supply a *ControllerControl* routine.

See Adapter Objects and DMA and Controller Objects for more information.

Handling I/O Control Requests in StartIo Routines

In general, only a subset of device I/O control requests are passed on from a driver's *DispatchDeviceControl* or *DispatchInternalDeviceControl* routine for further processing by the driver's *StartIo* routine. The driver's *StartIo* routine only has to handle valid device control requests that require device state changes or return volatile information about the current device state.

Each new driver must support the same set of public I/O control codes as all other drivers for the same kind of device. The system defines public, device-type-specific I/O control codes for **IRP_MJ_DEVICE_CONTROL**

requests as buffered requests.

Consequently, physical device drivers make data transfers to or from a system-space buffer that each driver finds in the IRP at **Irp->AssociatedIrp.SystemBuffer** for device control requests. Even drivers that set up their device objects for direct I/O use buffered I/O to satisfy device control requests with public I/O control codes.

The definition of each I/O control code determines whether data transferred for that request is buffered. Any privately defined I/O control codes for driver-specific **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests between paired drivers can define a code with method buffered, method direct, or method neither. As a general rule, any privately defined I/O control code should be defined with method neither if a closely coupled higher-level driver must allocate a buffer for that request.

Programming the Device for I/O Operations

Usually, the *StartIo* routine in a lowest-level device driver must synchronize access to any memory or device registers it shares with the driver's ISR by using **KeSynchronizeExecution** to call a driver-supplied *SynchCritSection* routine. The driver's *StartIo* routine uses the *SynchCritSection* routine to actually program the physical device for I/O at DIRQL. For more information, see Using Critical Sections.

Before calling **KeSynchronizeExecution**, the *StartIo* routine must do any preprocessing necessary for the request. Preprocessing might include calculating an initial partial-transfer range and saving any state information about the original request for other driver routines.

If a device driver uses DMA, its *StartIo* routine usually calls **AllocateAdapterChannel** with a driver-supplied *AdapterControl* routine. In these circumstances, the *StartIo* routine postpones the responsibility for programming the physical device to the *AdapterControl* routine. It, in turn, can call **KeSynchronizeExecution** to have a driver-supplied *SynchCritSection* routine program the device for a DMA transfer.

StartIo Routines in Higher-Level Drivers

6/25/2019 • 2 minutes to read • Edit Online

Any higher-level driver can have a *StartIo* routine. However, such a driver is unlikely to be interoperable with existing lower-level drivers and is likely to exhibit poor performance characteristics.

A *StartIo* routine in a higher-level driver has the following effects:

- Incoming IRPs can be queued by calling **IoStartPacket** from the driver's *Dispatch*Xxx routine(s) and **IoStartNextPacket** from its *IoCompletion* routine(s), thereby causing IRPs to be processed one at a time through the *StartIo* routine.
- The driver's I/O throughput could become noticeably slower during periods of heavy I/O demand, because its *StartIo* routine can become a bottleneck.
- The driver's *StartIo* routine calls **IoCallDriver** with each IRP at IRQL = DISPATCH_LEVEL, thereby causing all lower-level drivers' dispatch routines also to run at IRQL = DISPATCH_LEVEL. This restricts the set of support routines that lower drivers can call in their dispatch routines. Because most driver writers assume their drivers' dispatch routines run at IRQL < DISPATCH_LEVEL, the higher-level driver is unlikely to be interoperable with many existing lower-level drivers.
- The *StartIo* routine reduces overall system throughput because it and the dispatch routines of all lower-level drivers in its chain are run at IRQL = DISPATCH_LEVEL.

  For more information about the IRQLs at which standard driver routines are run, see Managing Hardware Priorities.

None of the system-supplied higher-level drivers has a *StartIo* routine, because it can slow IRP processing for the driver itself, for all drivers above and below it, and for the system overall.

Most higher-level drivers simply send IRPs to lower-level drivers from their dispatch routines and do any necessary clean-up processing in their *IoCompletion* routines.

However, higher-level drivers can set up internal queues for IRPs that request particular kinds of operations, or set up internal queues to hold IRPs bound for a set of heterogeneous underlying devices like the SCSI port driver. For more information, see Queuing and Dequeuing IRPs.

Points to Consider for StartIo Routines

6/25/2019 • 2 minutes to read • Edit Online

Keep the following points in mind when implementing a *StartIo* routine:

- A *StartIo* routine must synchronize its access to a physical device and to any shared state information or resources that the driver maintains in the device extension with the driver's other routines that access the same device, memory location, or resources.

  If the *StartIo* routine shares the device or state with the ISR, it must use **KeSynchronizeExecution** to call a driver-supplied *SynchCritSection* routine to program the device or to access the shared state. For more information, see Using Critical Sections.

  If the *StartIo* routine shares state or resources with routines other than the ISR, it must protect the shared state or resources with a driver-initialized executive spin lock for which the driver provides the storage. For more information, see Spin Locks.

- If a monolithic non-WDM device driver sets up a controller object, its *StartIo* routine can use the controller object to synchronize operations through a shared physical device with attached (similar) devices.

  See Controller Objects for more information.

- Unless a closely coupled higher-level driver presplits large DMA transfer requests for its underlying device driver, the underlying device driver's *StartIo* routine must split large transfer requests into partial-transfer ranges and the driver must carry out a sequence of partial-transfer device operations. Each partial transfer must be sized to suit the capabilities of the hardware: either the capabilities of the driver's device or, for a subordinate DMA device, the capabilities of the system DMA controller, whichever has stricter constraints.

  See Adapter Objects and DMA for more information about using system or bus-master DMA.

- The *StartIo* routine of a driver that uses DMA must synchronize transfers using an adapter object.

- A *StartIo* routine is run at IRQL = DISPATCH_LEVEL, which restricts the set of support routines it can call.

  For example, a *StartIo* routine can neither access nor allocate pageable memory, and it cannot wait for a dispatcher object to be set to the signaled state. On the other hand, a *StartIo* routine can acquire and release a driver-allocated executive spin lock with **KeAcquireSpinLockAtDpcLevel** and **KeReleaseSpinLockFromDpcLevel**, which run faster than **KeAcquireSpinLock** and **KeReleaseSpinLock**.

  See Managing Hardware Priorities and Spin Locks for more information.

- If the driver holds IRPs in a cancelable state, its *StartIo* routine must check whether the input IRP has already been canceled before it begins any processing for that request on its device. For more information, see Canceling IRPs.

Driver-Managed IRP Queues

6/25/2019 • 2 minutes to read • Edit Online

Except for file system drivers, the I/O manager associates a device queue object (for queuing IRPs) with each device object that a driver creates.

Most device drivers call the I/O manager's support routines to use the associated device queue, which holds IRPs whenever device I/O requests for a target device object come in faster than the driver can process them to completion. With this technique, IRPs are queued to a driver-supplied *StartIo* routine.

For good performance, most intermediate drivers simply pass IRPs on to lower drivers as fast as they come in, so intermediate drivers almost never use the device queues associated with their respective device objects.

However, you can design a driver to manage internal queues of IRPs by explicitly setting up one or more device queues, interlocked queues, or cancel-safe queues. This approach can be particularly useful if the driver controls a device that overlaps I/O operations. For such a device, it can be difficult to manage concurrent processing of two or more IRPs for the same target device object using only a single queue.

The simplest way to build an internal queue is to use the cancel-safe IRP queue framework. You can implement the queuing mechanism of your choice in your driver. You can then use **IoCsqInitialize** to register a set of callback routines that handle IRP insertion and deletion, as well as locking and unlocking your queue. The cancel-safe IRP queue framework provides the **IoCsqInsertIrp**, **IoCsqRemoveIrp**, and **IoCsqRemoveNextIrp** routines that automatically use the callback routines to safely insert and remove IRPs from the driver's queue. The system also uses your callback routines to safely remove any IRPs that are canceled.

You also might opt to set up supplemental queues for IRPs in the driver of a device controller for a set of heterogeneous physical devices. For example, the SCSI port driver uses device queue objects for internal queues. This driver both has a *StartIo* routine and sets up device queue objects as supplemental queues, in addition to the device queue associated with the device object it creates to represent an HBA. The SCSI port driver uses its supplemental device queues to hold IRPs bound for particular logical units on the HBA-controlled SCSI bus(es).

The system floppy controller driver is an example of a driver that has no *StartIo* routine and uses an interlocked queue. This driver sets up a doubly linked interlocked queue into which and from which the driver and its device-dedicated thread insert and remove IRPs.

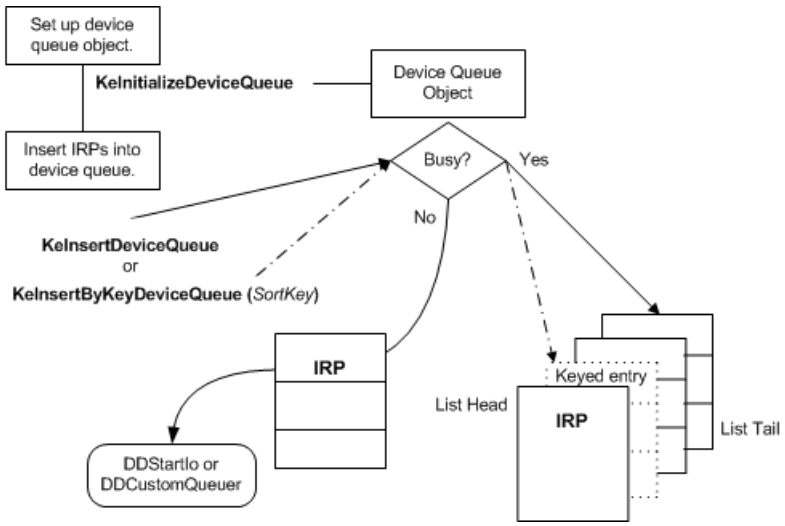
The Kernel defines the device queue object type. The executive support component provides routines for inserting and removing IRPs in interlocked queues. Drivers for Windows XP and later versions of Windows can use cancel-safe IRP queues to handle IRP queuing.

The following sections explain how to use device queues, interlocked queues, and cancel-safe queues:

Setting Up and Using Device Queues

6/25/2019 • 2 minutes to read • Edit Online

A driver sets up a device queue object by calling **KeInitializeDeviceQueue** at driver or device initialization. After starting its device(s), the driver inserts IRPs into this queue by calling **KeInsertDeviceQueue** or **KeInsertByKeyDeviceQueue**. The following figure illustrates these calls.



As this figure shows, the driver must provide the storage for a device queue object, which must be resident. Drivers that set up a device queue object usually provide the necessary storage in the device extension of a driver-created device object, but the storage can be in a controller extension if the driver uses a controller object or in nonpaged pool allocated by the driver.

If the driver provides storage for the device queue object in a device extension, it calls **KeInitializeDeviceQueue** after creating the device object and before starting the device. In other words, the driver can initialize the queue from its *AddDevice* routine or when it handles a PnP **IRP_MN_START_DEVICE** request. In the call to **KeInitializeDeviceQueue**, the driver passes a pointer to the storage it provides for the device queue object.

After starting its device(s), the driver can insert an IRP into its device queue by calling **KeInsertDeviceQueue**, which places the IRP at the tail of the queue, or **KeInsertByKeyDeviceQueue**, which places the IRP into the queue according to a driver-determined *SortKey* value, as shown in the previous figure.

Each of these support routines returns a Boolean value indicating whether the IRP was inserted into the queue. Each of these calls also sets the state of the device queue object to Busy if the queue is currently empty (Not-Busy). However, if the queue is empty (Not-Busy), neither **KeInsert*Xxx*DeviceQueue** routine inserts the IRP into the queue. Instead, it sets the state of the device queue object to Busy and returns **FALSE**. Because the IRP has not been queued, the driver must pass it on to another driver routine for further processing.

When setting up supplemental device queues, follow this implementation guideline:

When a call to **KeInsert*Xxx*DeviceQueue** returns **FALSE**, the caller must pass the IRP it attempted to queue on for further processing to another driver routine. However, the call to **KeInsert*Xxx*DeviceQueue** changes the state of the device queue object to Busy, so the next IRP to come in is inserted in the queue unless the driver calls **KeRemove*Xxx*DeviceQueue** first.

When the device queue object's state is set to Busy, the driver can dequeue an IRP for further processing or reset the state to Not-Busy by calling one of the following support routines:

- **KeRemoveDeviceQueue** to remove the IRP at the head of the queue

- **KeRemoveByKeyDeviceQueue** to remove an IRP chosen according to a driver-determined *SortKey* value
- **KeRemoveEntryDeviceQueue** to remove a particular IRP in the queue or to determine whether a particular IRP is in the queue

  KeRemoveEntryDeviceQueue returns a Boolean indicating whether the IRP was in the device queue.

Calling any of these routines to remove an entry from a device queue that is empty but Busy changes the queue state to Not-Busy.

Each device queue object is protected by a built-in executive spin lock (not shown in the Using a Device Queue Object figure). As a result, a driver can insert IRPs into the queue and remove them in a multiprocessor-safe manner from any driver routine running at less than or equal to IRQL = DISPATCH_LEVEL. Because of this IRQL restriction, a driver cannot call any **Ke*Xxx*DeviceQueue** routine from its ISR or *SynchCritSection* routines, which run at DIRQL.

See Managing Hardware Priorities and Spin Locks for more information. For IRQL requirements for a specific support routine, see the routine's reference page.

Managing Device Queues

6/25/2019 • 5 minutes to read • Edit Online

The I/O manager usually (except for FSDs) creates an associated device queue object when a driver calls **IoCreateDevice**. It also provides **IoStartPacket** and **IoStartNextPacket**, which drivers can call to have the I/O manager insert IRPs into the associated device queue or call their *StartIo* routines.

Consequently, it is rarely necessary (or particularly useful) for a driver to set up its own device queue objects for IRPs. Likely candidates are drivers, such as the SCSI port driver, that must coordinate incoming IRPs from some number of closely coupled class drivers for heterogeneous devices that are serviced through a single controller or bus adapter.

In other words, a driver for a disk array controller is more likely to use a driver-created controller object than to set up supplemental device queue object(s), while a driver for an add-on bus adapter and of a set of class drivers is slightly more likely to use supplemental device queues.

Using Supplemental Device Queues with a StartIo Routine

By calling **IoStartPacket** and **IoStartNextPacket**, a driver's Dispatch and *DpcForIsr* (or *CustomDpc*) routines synchronize calls to its *StartIo* routine using the device queue that the I/O manager created when the driver created the device object. For a port driver with a *StartIo* routine, **IoStartPacket** and **IoStartNextPacket** insert and remove IRPs in the device queue for the port driver's shared device controller/adapter. If the port driver also sets up supplemental device queues to hold requests coming in from closely coupled higher-level class drivers, it must "sort" incoming IRPs into its supplemental device queues, usually in its *StartIo* routine.

The port driver must determine which supplemental device queue each IRP belongs in before trying to insert that IRP into the appropriate queue. A pointer to the target device object is passed with the IRP to the driver's Dispatch routine. The driver should save the pointer for use in "sorting" the incoming IRPs. Note that the device object pointer passed to the *StartIo* routine is the driver's own device object, which represents the device controller/adapter, so it cannot be used for this purpose.

After queuing any IRPs, the driver programs its shared controller/adapter to carry out the request. Thus, the port driver can process incoming requests for all devices on a first-come, first-served basis until a call to **KeInsertDeviceQueue** puts an IRP into a particular class driver's device queue.

By using its own device queue for all IRPs to be processed through its *StartIo* routine, the underlying port driver serializes operations through the shared device (or bus) controller/adapter to all attached devices. By sometimes holding IRPs for each supported device in a separate device queue, this port driver inhibits the processing of IRPs for an already busy device while increasing I/O throughput for every other device that does I/O through its shared hardware.

In response to the call to **IoStartPacket** from the port driver's Dispatch routine, the I/O manager either calls that driver's *StartIo* routine immediately or puts the IRP into the device queue associated with the device object for the port driver's shared controller/adapter.

The port driver must maintain its own state information about each of the heterogeneous devices that it services through the shared device controller/adapter.

Keep in mind the following when designing class/port drivers with supplemental device queues:

- A driver cannot easily get a pointer to a device object created by any driver layered above itself, except for the device object at the top of its device stack.

  By design, the I/O manager does not provide a support routine for getting such a pointer. Moreover, the

order in which drivers are loaded makes it impossible for lower drivers to get pointers for higher-level drivers' device objects, which have not yet been created when any lower-level driver is adding its device.

Although **IoGetAttachedDeviceReference** returns a pointer to the highest-level device object in a driver's stack, a driver should use this pointer only to designate a target for I/O requests to its stack. A driver should not attempt to read or write the device object.

- A driver cannot use a pointer to a device object created by any driver layered above itself, except to send requests to the top of its own device stack.

  There is no way to synchronize access to a single device object (and its device extension) between two drivers in a multiprocessor-safe manner. Neither driver can make any assumptions about what I/O processing the other driver is currently doing.

Even for closely coupled class/port drivers, each class driver should use the pointer to the port driver's device object(s) only to pass on IRPs using **IoCallDriver**. The underlying port driver must maintain its own state, probably in the port driver's device extension, about requests that it processes for any closely coupled class driver(s)' device(s).

Managing Supplemental Device Queues Across Driver Routines

Any port driver that queues IRPs in supplemental device queues for a closely coupled set of class drivers also must handle the following situation efficiently:

1. Its Dispatch routines have inserted IRPs for a particular device in the driver-created device queue for that device.
2. IRPs for other devices continue to come in, to be queued to the driver's *StartIo* routine with **IoStartPacket**, and to be processed through the shared device controller.
3. The device controller does not become idle, but each IRP held in the driver-created device queue also must be queued to the driver's *StartIo* routine as soon as possible.

Consequently, the port driver's *DpcForIsr* routine must attempt to transfer an IRP from the driver's internal device queue for a particular device into the device queue for the shared adapter/controller whenever the port driver completes an IRP, as follows:

1. The *DpcForIsr* routine calls **IoStartNextPacket** to have the *StartIo* routine begin processing the next IRP queued to the shared device controller.
2. The *DpcForIsr* routine calls **KeRemoveDeviceQueue** to dequeue the next IRP (if any) that it is holding in its internal device queue for the device on whose behalf it is about to complete an IRP.
3. If **KeRemoveDeviceQueue** returns a non-NULL pointer, the *DpcForIsr* routine calls **IoStartPacket** with the just dequeued IRP to have it queued to the shared device controller/adapter. Otherwise, the call to **KeRemoveDeviceQueue** simply resets the state of the device queue object to Not-Busy, and the *DpcForIsr* routine omits the call to **IoStartPacket**.
4. Then, the *DpcForIsr* routine calls **IoCompleteRequest** with the input IRP for which the port driver has just completed I/O processing, either by setting the I/O status block with an error or by satisfying the I/O request.

Note that the preceding sequence implies that the *DpcForIsr* routine also must determine the device for which it is completing the current (input) IRP in order to manage internal queuing of IRPs efficiently.

If the port driver attempts to wait until its shared controller/adapter is idle before dequeuing IRPs held in its supplemental device queues, the driver might starve a device for which there was heavy I/O demand while it promptly serviced every other device for which the current I/O demand was actually much lighter.

Setting Up and Using Interlocked Queues

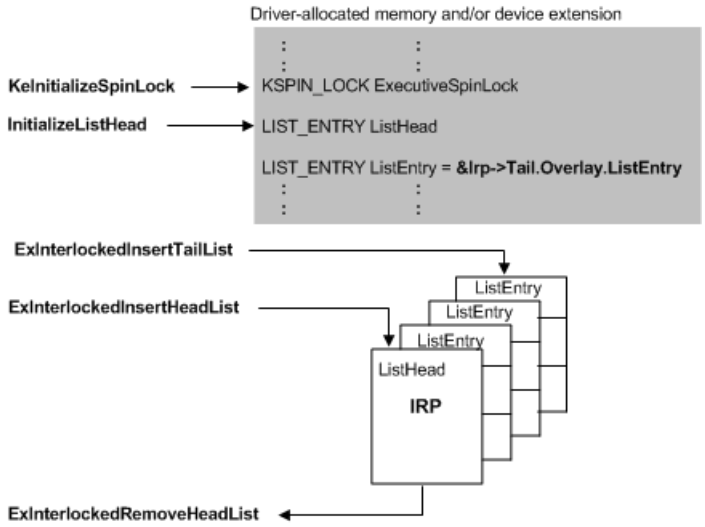
6/25/2019 • 3 minutes to read • Edit Online

New drivers should use the cancel-safe IRP queue framework in preference to the methods outlined in this section.

Drivers with device-dedicated threads or drivers that use executive worker threads, such as most system FSDs, are the most likely types of drivers to manage their own run-time internal queuing of IRPs in an interlocked queue. All PnP drivers, including WDM drivers, also must queue certain IRPs internally while making PnP and power state transitions.

Usually, these drivers set up a doubly linked interlocked queue; every IRP contains a member of type **LIST_ENTRY**, which a driver can use to doubly link IRPs that it is currently holding. A driver cannot requeue IRPs for retries if it sets up a singly linked interlocked queue.

A driver must set up its interlocked queue at device initialization. The following figure illustrates a doubly linked interlocked queue, the support routines a driver must call to set up such a queue, and a set of **ExInterlocked*Xxx*** routines a driver can call to insert IRPs into and remove IRPs from the queue.



As this figure shows, a driver must provide the storage for the queue itself and for the following in order to set up a doubly linked interlocked queue:

- An executive spin lock, which the driver must call **KeInitializeSpinLock** to initialize. Usually, a driver initializes the spin lock when it sets up the device extension(s) for its device object(s) in its *AddDevice* routine.
- The list head for the queue, which the driver must initialize by calling **InitializeListHead**.

Most drivers that use doubly linked interlocked queues provide the necessary storage in the device extension of a driver-created device object. The queue and executive spin lock can instead be in a controller extension (if the driver uses a controller object) or in nonpaged pool allocated by the driver.

While the driver is accepting I/O requests, it can insert an IRP into its queue by calling either of the following support routines if the *ListHead* is of type **LIST_ENTRY**, as shown in the previous figure:

ExInterlockedInsertTailList to place the IRP at the end of the queue

ExInterlockedInsertHeadList to place the IRP at the front of the queue. Drivers usually call this routine only when they must retry a particular request.

The driver must pass pointers to the IRP (*ListEntry*), as well the *ListHead* and executive spin lock (*Lock*) pointers that it previously initialized, to each of these **ExInterlockedInsert*Xxx*List** routines. Only pointers to the *ListHead* and *Lock* are required when the driver dequeues an IRP by calling **ExInterlockedRemoveHeadList**. To prevent deadlocks, the driver must not be holding an ExecutiveSpinLock that it passes to any **ExInterlocked*Xxx*** routine.

Because an interlocked queue is protected by the executive spin lock, the driver can insert IRPs into its doubly linked queue and remove them in a multiprocessor-safe manner from any driver routine running at less than or equal to IRQL = DISPATCH_LEVEL.

A queue with a ListHead of type **LIST_ENTRY**, as shown in the previous figure, is a doubly linked list. One with a ListHead of type **SLIST_HEADER** is a sequenced, singly linked list. A driver initializes the ListHead for a sequenced singly linked interlocked queue by calling **ExInitializeSListHead**.

A driver that never retries I/O operations can use **ExInterlockedPushEntrySList** and **ExInterlockedPopEntrySList** to manage its queuing of IRPs internally in a sequenced, singly linked interlocked queue. Any driver that uses this type of interlocked queue also must provide resident storage for a ListHead of type **SLIST_HEADER** and for an ExecutiveSpinLock, as shown in the previous figure. It must initialize the spin lock and set up its queue before calling **ExInterlockedPushEntrySList** to insert the initial entry into its queue.

For more information, see Managing Hardware Priorities and Spin Locks. For IRQL requirements for a specific support routine, see the routine's reference page.

Managing Interlocked Queues with a Driver-Created Thread

6/25/2019 • 3 minutes to read • Edit Online

New drivers should use the cancel-safe IRP queue framework in preference to the methods outlined in this section.

Like the system floppy controller driver, a driver with a device-dedicated thread, rather than a *StartIo* routine, usually manages its own queuing of IRPs in a doubly linked interlocked queue. The driver's thread pulls IRPs from its interlocked queue when there is work to be done on the device.

In general, the driver must manage synchronization with its thread to any resources shared between the thread and other driver routines. The driver also must have some way to notify its driver-created thread that IRPs are queued. Usually, the thread waits on a dispatcher object, stored in the device extension, until the driver's Dispatch routines set the dispatcher object to the Signaled state after inserting an IRP into the interlocked queue.

When the driver's Dispatch routines are called, each checks the parameters in the I/O stack location of the input IRP and, if they are valid, queues the request for further processing. For each IRP queued to a driver-dedicated thread, the dispatch routine should set up whatever context its thread needs to process that IRP before it calls **ExInterlockedInsert*Xxx*List**. The driver's I/O stack location in each IRP gives the driver's thread access to the device extension of the target device object, where the driver can share context information with its thread, as the thread removes each IRP from the queue.

A driver that queue cancelable IRPs must implement a *Cancel* routine. Since IRPs are canceled asynchronously, you must ensure that your driver avoids the race conditions that can result. See Synchronizing IRP Cancellation For more information about race conditions associated with canceling IRPs and techniques to avoid them.

Any driver-created thread runs at IRQL = PASSIVE_LEVEL and at a base run-time priority previously set when the driver called **PsCreateSystemThread**. The thread's call to **ExInterlockedRemoveHeadList** temporarily raises the IRQL to DISPATCH_LEVEL on the current processor while the IRP is being removed from the driver's internal queue. The original IRQL is restored to PASSIVE_LEVEL on return from this call.

Any driver thread (or driver-supplied worker-thread callback) must carefully manage the IRQLs at which it runs. For example, consider the following:

- Because system threads generally run at IRQL = PASSIVE_LEVEL, it is possible for a driver thread to wait for kernel-defined dispatcher objects to be set to the signaled state.

  For example, a device-dedicated thread might wait for other drivers to satisfy an event and complete some number of partial-transfer IRPs that the thread sets up with **IoBuildSynchronousFsdRequest**.

- However, such a device-dedicated thread must raise IRQL on the current processor before it calls certain support routines.

  For example, if a driver uses DMA, its device-dedicated thread must nest its calls to **AllocateAdapterChannel** and **FreeAdapterChannel** between calls to **KeRaiseIrql** and **KeLowerIrql** because these routines and certain other support routines for DMA operations must be called at IRQL = DISPATCH_LEVEL.

  Remember that *StartIo* routines are run at DISPATCH_LEVEL, so drivers that use DMA need not make calls to the **Ke*Xxx*Irql** routines from their *StartIo* routines.

- A driver-created thread can access pageable memory because it runs in a nonarbitrary thread context (its

own) at IRQL = PASSIVE_LEVEL, but many other standard driver routines run at IRQL >= DISPATCH_LEVEL. If a driver-created thread allocates memory that can be accessed by such a routine, it must allocate the memory from nonpaged pool. For example, if a device-dedicated thread allocates any buffer that will be accessed later by the driver's ISR or *SynchCritSection*, *AdapterControl*, *AdapterListControl*, *ControllerControl*, *DpcForIsr*, *CustomDpc*, *IoTimer*, *CustomTimerDpc*, or, in a higher-level driver, *IoCompletion* routine, the thread-allocated memory cannot be pageable.

- If the driver maintains shared state information or resources in a device extension, a driver thread (like a *StartIo* routine) must synchronize its access to a physical device and to the shared data with the driver's other routines that access the same device, memory location, or resources.

  If the thread shares the device or state with the ISR, it must use **KeSynchronizeExecution** to call a driver-supplied *SynchCritSection* routine to program the device or to access the shared state. See Using Critical Sections.

  If the thread shares state or resources with routines other than the ISR, the driver must protect the shared state or resources with a driver-initialized executive spin lock for which the driver provides the storage. For more information, see Spin Locks.

For more information about the design tradeoffs of a using a driver thread for a slow device, see Polling a Device. See also Managing Hardware Priorities. For specific information about IRQLs for particular support routines, see the routine's reference page.

Cancel-Safe IRP Queues

6/25/2019 • 6 minutes to read • Edit Online

Drivers that implement their own IRP queuing should use the *cancel-safe IRP queue* framework. Cancel-safe IRP queues split IRP handling into two parts:

1. The driver provides a set of callback routines that implement standard operations on the driver's IRP queue. The provided operations include inserting and removing IRPs from the queue, and locking and unlocking the queue. See Implementing the Cancel-Safe IRP Queue.
2. Whenever the driver needs to actually insert or remove an IRP from the queue, it uses the system-provided **IoCsq*Xxx*** routines. These routines handle all synchronization and IRP canceling logic for the driver.

Drivers that use cancel-safe IRP queues do not implement *Cancel* routines to support IRP cancellation.

The framework ensures that drivers insert and remove IRPs from their queue atomically. It also ensures that IRP cancellation is implemented correctly. Drivers that do not use the framework must manually lock and unlock the queue before performing any insertions and deletions. They must also avoid the race conditions that can result when implementing a *Cancel* routine. (For a description of the race conditions that can arise, see Synchronizing IRP Cancellation.)

The cancel-safe IRP queue framework is included with Windows XP and later versions of Windows. Drivers that must also work with Windows 2000 and Windows 98/Me can link to the Csq.lib library that is included in the Windows Driver Kit (WDK). The Csq.lib library provides an implementation of this framework.

The **IoCsq*Xxx*** routines are declared in the Windows XP and later versions of Wdm.h and Ntddk.h. Drivers that must also work with Windows 2000 and Windows 98/Me must include Csq.h for the declarations.

You can see a complete demonstration of how to use cancel-safe IRP queues in the \src\general\cancel directory of the WDK. For more information about these queues, also see the Flow of Control for Cancel-Safe IRP Queuing white paper.

Implementing the Cancel-Safe IRP Queue

To implement a cancel-safe IRP queue, drivers must provide the following routines:

- Either of the following routines to insert IRPs into the queue: *CsqInsertIrp* or *CsqInsertIrpEx*. *CsqInsertIrpEx* is an extended version of *CsqInsertIrp*; the queue is implemented using one or the other.
- A *CsqRemoveIrp* routine that removes the specified IRP from the queue.
- A *CsqPeekNextIrp* routine that returns a pointer to the next IRP following the specified IRP in the queue. This is where the system passes the *PeekContext* value that it receives from **IoCsqRemoveNextIrp**. The driver can interpret that value in any way.
- Both of the following routines to allow the system to lock and unlock the IRP queue: *CsqAcquireLock* and *CsqReleaseLock*.
- A *CsqCompleteCanceledIrp* routine that completes a canceled IRP.

Pointers to the driver's routines are stored in the **IO_CSQ** structure that describes the queue. The driver allocates the storage for the **IO_CSQ** structure. The **IO_CSQ** structure is guaranteed to remain a fixed size, so a driver can safely embed the structure inside its device extension.

The driver uses either **IoCsqInitialize** or **IoCsqInitializeEx** to initialize the structure. Use **IoCsqInitialize** if

the queue implements *CsqInsertIrp*, or **IoCsqInitializeEx** if the queue implements *CsqInsertIrpEx*.

Drivers need only provide the essential functionality in each callback routine. For example, only the *CsqAcquireLock* and *CsqReleaseLock* routines implement lock handling. The system automatically calls these routines to lock and unlock the queue as necessary.

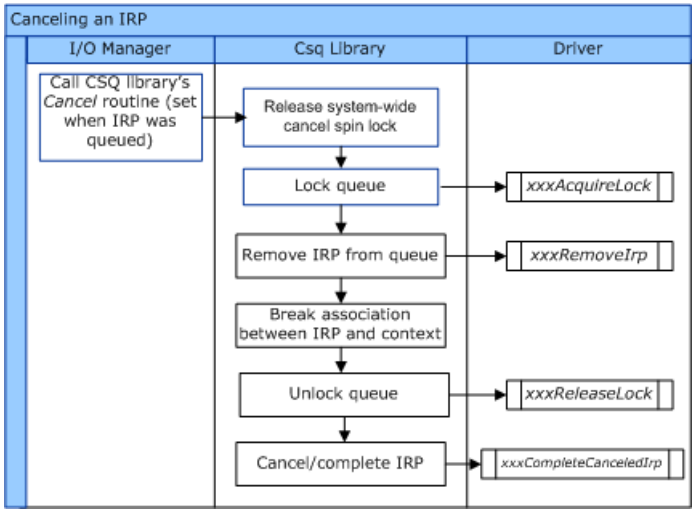
You can implement any type of IRP queuing mechanism in your driver, as long as the appropriate dispatch routines are provided. For example, the driver could implement the queue as a linked list, or as a priority queue.

CsqInsertIrpEx provides a more flexible interface to the queue than does *CsqInsertIrp*. The driver can use its return value to indicate the result of the operation; if it returns an error code, the insertion failed. A *CsqInsertIrp* routine does not return a value, so there is no simple way to indicate that an insertion failed. Also, *CsqInsertIrpEx* takes an additional driver-defined *InsertContext* parameter that can be used to specify additional driver-specific information to be used by the queue implementation.

Drivers can use *CsqInsertIrpEx* to implement more sophisticated IRP handling. For example, if there are no pending IRPs, the *CsqInsertIrpEx* routine can return an error code and the driver can process the IRP immediately. Similarly, if IRPs can no longer be queued, the *CsqInsertIrpEx* can return an error code to indicate that fact.

The driver is insulated from all IRP cancellation handling. The system provides a *Cancel* routine for IRPs in the queue. This routine calls *CsqRemoveIrp* to remove the IRP from the queue, and *CsqCompleteCanceledIrp* to complete the IRP cancellation.

The following diagram illustrates the flow of control for IRP cancellation.



A basic implementation of *CsqCompleteCanceledIrp* is as follows.

```
VOID CsqCompleteCanceledIrp(PIO_CSQ Csq, PIRP Irp) {
  Irp->IoStatus.Status = STATUS_CANCELLED;
  Irp->IoStatus.Information = 0;

  IoCompleteRequest(Irp, IO_NO_INCREMENT);
}
```

Drivers can use any of the operating system's synchronization primitives to implement their *CsqAcquireLock* and *CsqReleaseLock* routines. Available synchronization primitives include spin locks and mutex objects.

Here is an example of how a driver can implement locking using spin locks.

```
/*
  The driver has previously initialized the SpinLock variable with
  KeInitializeSpinLock.
 */

VOID CsqAcquireLock(PIO_CSQ IoCsq, PKIRQL PIrql)
{
    KeAcquireSpinLock(SpinLock, PIrql);
}

VOID CsqReleaseLock(PIO_CSQ IoCsq, KIRQL Irql)
{
    KeReleaseSpinLock(SpinLock, Irql);
}
```

The system passes a pointer to an IRQL variable to *CsqAcquireLock* and *CsqReleaseLock*. If the driver uses a spin lock to implement locking for the queue, the driver can use this variable to store the current IRQL when the queue is locked.

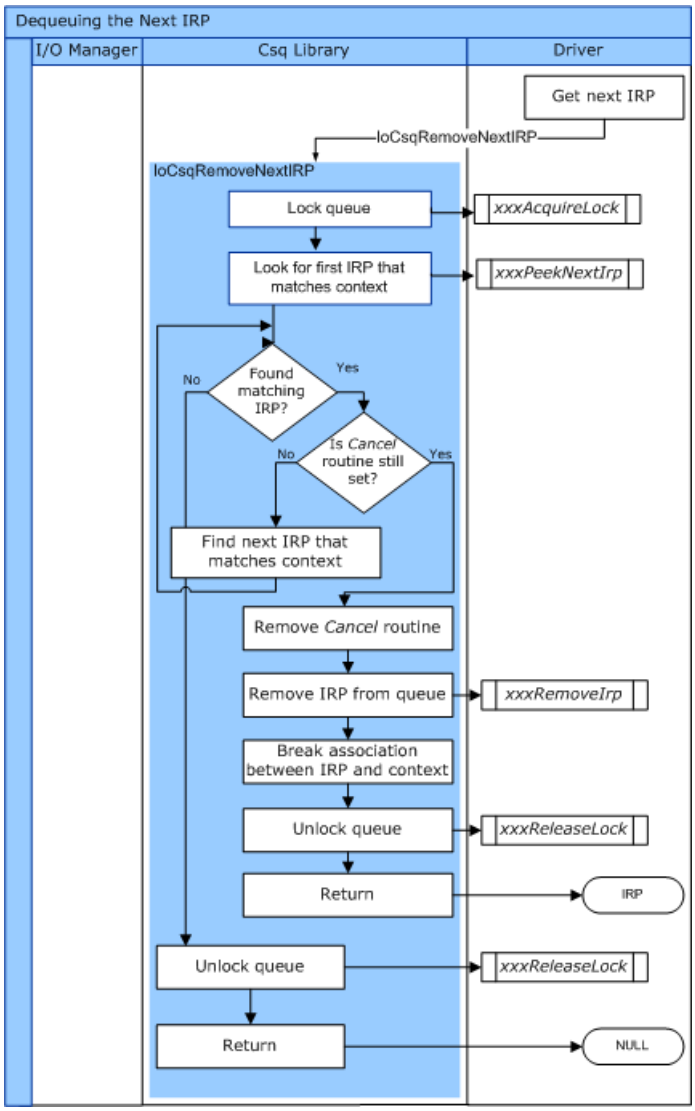
Drivers are not required to use spin locks. For example, the driver could use a mutex to lock the queue. For a description of the synchronization techniques that are available to drivers, see Synchronization Techniques.

Using the Cancel-Safe IRP Queue

Drivers use the following system routines when queuing and dequeuing IRPs:

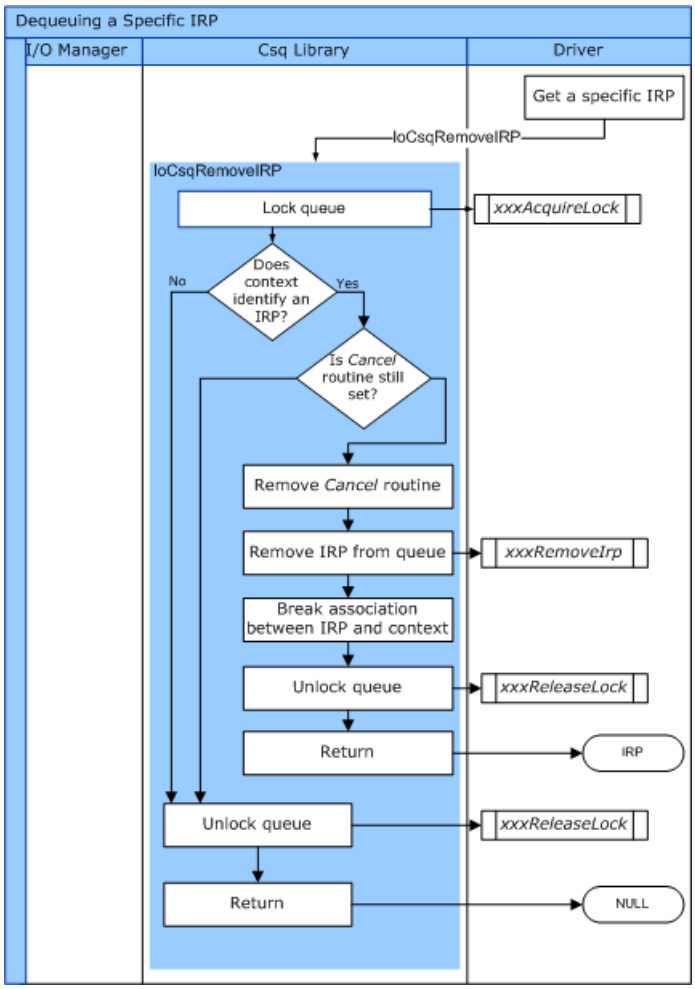
- Either of the following to insert an IRP into the queue: **IoCsqInsertIrp** or **IoCsqInsertIrpEx**.
- **IoCsqRemoveNextIrp** to remove the next IRP in the queue. The driver can optionally specify a key value.

The following diagram illustrates the flow of control for **IoCsqRemoveNextIrp**.



- **IoCsqRemoveIrp** to remove the specified IRP from the queue.

The following diagram illustrates the flow of control for **IoCsqRemoveIrp**.

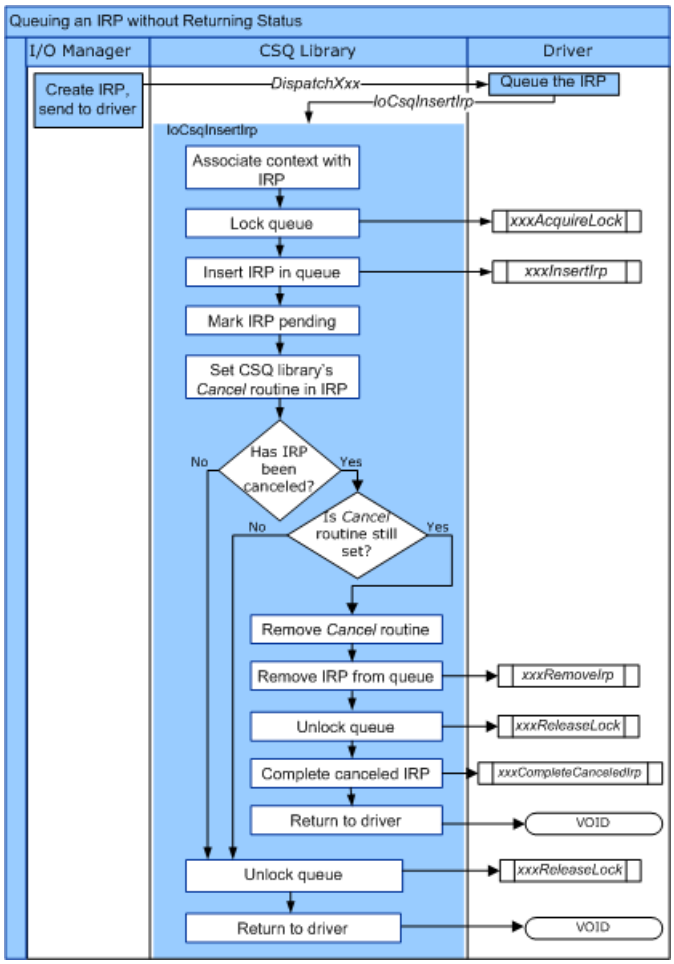


These routines, in turn, dispatch to driver-supplied routines.

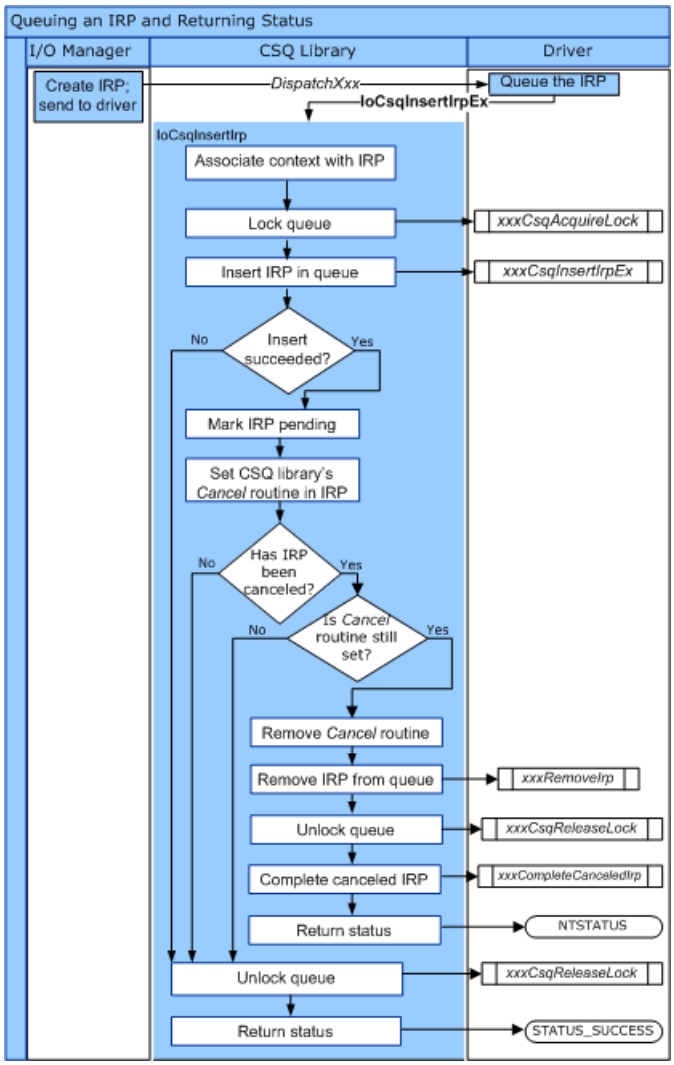
The **IoCsqInsertIrpEx** routine provides access to the extended features of a *CsqInsertIrpEx* routine. It returns the status value that was returned by *CsqInsertIrpEx*. The caller can use this value to determine if the IRP was successfully queued or not. **IoCsqInsertIrpEx** also allows the caller to specify a value for the InsertContext parameter of *CsqInsertIrpEx*.

Note that both **IoCsqInsertIrp** and **IoCsqInsertIrpEx** can be called on any cancel-safe queue, whether the queue has a *CsqInsertIrp* routine or a *CsqInsertIrpEx* routine. **IoCsqInsertIrp** behaves the same in either case. If **IoCsqInsertIrpEx** is passed a queue that has a *CsqInsertIrp* routine, it behaves identically to **IoCsqInsertIrp**.

The following diagram illustrates the flow of control for **IoCsqInsertIrp**.



The following diagram illustrates the flow of control for **IoCsqInsertIrpEx**.



There are several natural ways to use the **IoCsq*Xxx*** routines to queue and dequeue IRPs. For example, a driver could simply queue IRPs to be processed in the order in which they are received. The driver could queue an IRP as follows:

```
status = IoCsqInsertIrpEx(IoCsq, Irp, NULL, NULL);
```

If the driver is not required to distinguish between particular IRPs, it could then simply dequeue them in the order in which they were queued, as follows:

```
IoCsqRemoveNextIrp(IoCsq, NULL);
```

Alternatively, the driver could queue and dequeue specific IRPs. The routines use the opaque **IO_CSQ_IRP_CONTEXT** structure to identify particular IRPs in the queue. The driver queues the IRP as follows:

```
IO_CSQ_IRP_CONTEXT ParticularIrpInQueue;
IoCsqInsertIrp(IoCsq, Irp, &ParticularIrpInQueue);
```

The driver can then dequeue the same IRP by using the **IO_CSQ_IRP_CONTEXT** value.

```
IoCsqRemoveIrp(IoCsq, Irp, &ParticularIrpInQueue);
```

The driver might also be required to remove IRPs from the queue based on a particular criterion. For example,

the driver might associate a priority with each IRP, such that higher priority IRPs get dequeued first. The driver might pass a *PeekContext* value to **IoCsqRemoveNextIrp**, which the system passes back to the driver when it requests the next IRP in the queue.

Completing IRPs

6/25/2019 • 2 minutes to read • Edit Online

"Completing an IRP" is a shorthand phrase that means "allowing all members of the driver stack to complete an I/O operation." After the IRP has been completed, the I/O manager notifies the initiating application that the requested I/O operation has finished.

When a driver has finished processing an IRP, it calls **IoCompleteRequest** (typically from within a *DpcForIsr* routine). This causes the I/O manager to determine whether any higher-level drivers have set up *IoCompletion* routines for the IRP. If so, each *IoCompletion* routine is called, in turn, until every layered driver in the chain has completed the IRP.

When all drivers have completed the IRP, the I/O manager returns status to the original requester of the operation. Note that a higher-level driver that sets up a driver-created IRP must supply an *IoCompletion* routine to release the IRP it created.

This section contains the following topics:

When to Complete an IRP

6/25/2019 • 2 minutes to read • Edit Online

A driver should initiate IRP completion when any of the following conditions is met:

- The driver determines that IRP processing cannot progress because of invalid parameters or other conditions.
- The driver is able to handle the requested I/O operation without passing the IRP down the driver stack, and the operation has finished.
- The IRP is being canceled. (See Canceling IRPs.)

If these conditions are not met, a driver's dispatch routine must pass the IRP down to the next-lower driver, or it must handle processing of the I/O request. If one of the conditions is met, the driver must call **IoCompleteRequest**.

If a driver completes a request because processing cannot progress, or if it completes a request by handling the requested operation without actually accessing the device, it typically calls **IoCompleteRequest** from one of its dispatch routines. For more information, see Completing IRPs in Dispatch Routines.

If a driver must access a device to satisfy the request, it typically calls **IoCompleteRequest** from a *DpcForIsr* routine. These routines are discussed extensively in Servicing Interrupts.

How to Complete an IRP in a Dispatch Routine

6/25/2019 • 2 minutes to read • Edit Online

If an input IRP can be completed immediately, a dispatch routine does the following:

1. Sets the **Status** and **Information** members of the IRP's I/O status block with appropriate values, in general:

   - The dispatch routine sets **Status** either to STATUS_SUCCESS or to an appropriate error (STATUS_*XXX*), which can be the value returned by a call to a support routine or, for certain synchronous requests, by a lower driver.

     If a lower-level driver returns STATUS_PENDING, a higher-level driver should not call **IoCompleteRequest** for the IRP, with one exception: The higher-level driver can use an event to synchronize between its *IoCompletion* routine and its dispatch routine, in which case the *IoCompletion* routine signals the event and returns STATUS_MORE_PROCESSING_REQUIRED. The dispatch routine waits for the event and then calls **IoCompleteRequest** to complete the IRP.

   - It sets **Information** to the number of bytes successfully transferred if a request to transfer data, such as a read or write request, was satisfied.

   - It sets **Information** to a value that varies according to the specific request for other IRPs that it completes with STATUS_SUCCESS.

   - It sets **Information** to a value that varies according to the specific request for IRPs that it completes with a warning STATUS_*XXX*. For example, it would set **Information** to the number of bytes transferred for such a warning as STATUS_BUFFER_OVERFLOW.

   - Usually, it sets **Information** to zero for requests that it completes with an error STATUS_*XXX*.

2. Calls **IoCompleteRequest** with the IRP and with *PriorityBoost* = IO_NO_INCREMENT.

3. Returns the appropriate STATUS_*XXX* that it already set in the I/O status block. Note that a call to **IoCompleteRequest** makes the given IRP inaccessible by the caller, so the return value from a dispatch routine cannot be set from the I/O status block of an already completed IRP.

Follow this implementation guideline for calling IoCompleteRequest with an IRP:

Always release any spin lock(s) the driver is holding before calling **IoCompleteRequest**.

It takes an indeterminate amount of time to complete an IRP, particularly in a chain of layered drivers. Moreover, a deadlock can occur if a higher-level driver's *IoCompletion* routine sends an IRP back down to a lower driver that is holding a spin lock.

When to Complete an IRP in a Dispatch Routine

7/9/2019 • 2 minutes to read • Edit Online

Usually, drivers do not complete IRPs in their dispatch routines unless the parameters for the given request are invalid or, in a device driver, unless the particular **IRP_MJ_*XXX*** requires no device I/O operations.

Every driver in a chain of layered drivers can check the validity of parameters in its own I/O stack location, for each IRP received by the driver's dispatch routines. Completing IRPs with invalid parameters in the dispatch routine of the highest possible driver improves I/O throughput for any chain of drivers and for the system overall.

A dispatch routine in a higher-level driver should either complete an IRP or pass it on for processing by lower drivers, according to the following guidelines:

- If the dispatch routine determines that any parameters in its own I/O stack location are invalid, it should complete that IRP immediately with an appropriate error status, such as STATUS_INVALID_PARAMETER.
- If the IRP contains the function code **IRP_MJ_CLEANUP**, the *DispatchCleanup* routine must complete every IRP currently queued to the target device object, for the file object specified in the driver's I/O stack location, and complete the cleanup IRP.

  A cleanup request indicates that an application is being terminated or has closed a file handle for the file object that represents the driver's device object. When the *DispatchCleanup* routine returns, usually the driver's *DispatchClose* routine is called next.
- Otherwise, a higher-level driver can satisfy the request only by passing it on to the next-lower driver.

A dispatch routine in a lowest-level driver should complete an IRP according to the following guidelines:

- If the dispatch routine determines that any parameters in its own I/O stack location are invalid, or if the driver does not support the IRP, it should complete that IRP immediately with an appropriate error status. In such cases the driver must not complete the IRP with a status value of STATUS_SUCCESS.

  Usually, any higher-level driver has already checked the parameters for a requested operation, but lowest-level device drivers should perform their own parameters checks as well.
- If the IRP contains the function code **IRP_MJ_CLEANUP**, the *DispatchCleanup* routine must complete every IRP currently queued to the target device object, for the given file object in the driver's I/O stack location, and then complete the cleanup IRP.

  A cleanup request indicates that an application is being terminated or has closed a file handle for the file object that represents the driver's device object. When the *DispatchCleanup* routine returns, usually the driver's *DispatchClose* routine is called next.
- If the request requires no device I/O operation, the dispatch routine should satisfy the request and complete the IRP.

  For example, a driver might save the current mode of its device in the device extension, particularly if it seldom changes device modes after initialization. Its *DispatchDeviceControl* routine could then satisfy a request that queried the current device mode by returning this stored information.

Otherwise, the dispatch routine must call **IoMarkIrpPending**, queue the IRP to other driver routines for further processing, and return STATUS_PENDING.

Using IoCompletion Routines

6/25/2019 • 2 minutes to read • Edit Online

Higher-level drivers that monitor on an IRP-specific basis how lower-level drivers carried out particular requests can have one or more *IoCompletion* routines. Higher-level drivers that allocate IRPs to send requests to lower drivers must have an *IoCompletion* routine.

A highest-level or intermediate driver's *DispatchRead* or *DispatchWrite* routine is most likely to set an *IoCompletion* routine for an IRP, because lower-level drivers must handle transfer requests asynchronously.

The lowest-level driver in a driver stack cannot register *IoCompletion* routines.

Drivers generally do not register *IoCompletion* routines for IRPs associated with synchronous I/O operations. For instance, a higher-level driver's *DispatchDeviceControl* routine can allocate an IRP using **IoBuildDeviceIoControlRequest**. In this case, the dispatch routine typically does not register an *IoCompletion* routine, because device control requests are generally handled synchronously. Instead, the driver can allocate and initialize an event object, and its *DispatchDeviceControl* routine can wait for an event to be initialized when it sends on driver-allocated IRPs. Usually, a higher-level driver does not register an *IoCompletion* routine for an IRP allocated with **IoBuildSynchronousFsdRequest**, for the same reason.

This section contains the following topics:

Registering an IoCompletion Routine

6/25/2019 • 3 minutes to read • Edit Online

To register an *IoCompletion* routine, a dispatch routine calls **IoSetCompletionRoutine**, supplying the *IoCompletion* routine's address and the IRP that it will subsequently pass on to lower drivers using **IoCallDriver**.

When it calls **IoSetCompletionRoutine**, the dispatch routine specifies the circumstances in which the I/O manager should call the specified *IoCompletion* routine. You can choose to have the *IoCompletion* routine called if a lower level driver completes the IRP successfully (*InvokeOnSuccess*), completes the IRP with an error status value (*InvokeOnError*), or cancels the IRP (*InvokeOnCancel*), in any combination.

The purpose of an *IoCompletion* routine is to monitor what lower-level drivers did with the IRP and to do additional completion processing, if necessary. Specifically, the most common uses for a driver's *IoCompletion* routines are the following:

- To dispose of an IRP that the driver allocated with **IoAllocateIrp** or **IoBuildAsynchronousFsdRequest**

  Any higher-level driver that allocates an IRP using either of these support routines must supply an *IoCompletion* routine for that IRP. The *IoCompletion* routine must call **IoFreeIrp** to dispose of driver-allocated IRPs.

- To reuse an incoming IRP to request that lower drivers complete some number of operations, such as partial transfers, until the original request can be satisfied and completed by the *IoCompletion* routine

- To retry a request that a lower driver completed with an error

  Highest-level drivers, such as file systems, are more likely to have *IoCompletion* routines that attempt to retry requests than are intermediate drivers, except possibly class drivers layered above a closely coupled port driver. However, any intermediate driver use *IoCompletion* routines to retry requests.

While a highest-level or intermediate driver's *DispatchReadWrite* routine is most likely to process IRPs that require an *IoCompletion* routine, any dispatch routine in any driver that passes IRPs on to lower drivers can register an *IoCompletion* routine.

For driver-allocated IRPs and reused IRPs, the dispatch routine must call **IoSetCompletionRoutine** with the following Boolean parameters:

- *InvokeOnSuccess* set to **TRUE**
- *InvokeOnError* set to **TRUE**
- *InvokeOnCancel* set to **TRUE** if any lower driver in the chain might handle cancelable IRPs

  Usually, *InvokeOnCancel* is set to **TRUE**, regardless of whether an IRP might be returned with STATUS_CANCELLED, to ensure that the *IoCompletion* routine frees each driver-allocated IRP or checks the completion status of each reuse of an IRP.

A dispatch routine that allocates IRPs for lower drivers using **IoAllocateIrp** or **IoBuildAsynchronousFsdRequest**must set an *IoCompletion* routine for each driver-allocated IRP.

- The dispatch routine must set up state about both the original IRP and its allocated IRP(s) for the *IoCompletion* routine to use. At a minimum, the *IoCompletion* routine needs access to the original IRP and a count of how many additional IRPs were allocated.
- The dispatch routine should call **IoSetCompletionRoutine** with all *InvokeOnXxx* parameters set to **TRUE**

for the IRP(s) it allocates.

A dispatch routine that reuses IRPs for a sequence of operations, or that retries I/O operation, must call **IoSetCompletionRoutine** for each IRP that will be reused or retried.

- The dispatch routine must save the original IRP's state information, for subsequent use by the *IoCompletion* routine.

  For example, a *DispatchReadWrite* routine must save the relevant transfer parameters of an input IRP for the *IoCompletion* routine before setting up a partial transfer for the next-lower driver in that IRP. Saving the parameters is particularly important if the *DispatchReadWrite* routine modifies any parameters that the *IoCompletion* routine needs to determine when the original request has been satisfied.

- If the *IoCompletion* routine can retry the request, the dispatch routine must set up a driver-determined upper limit for the number of retries its *IoCompletion* routine should attempt before it completes the original IRP with an error.

- If an IRP is to be reused, the dispatch routine should call **IoSetCompletionRoutine** with all *InvokeOnXxx* parameters set to **TRUE**.

- For an asynchronous request, the dispatch routine of any intermediate driver must call **IoMarkIrpPending** for the original IRP. It must then return STATUS_PENDING after it has sent the IRP on to lower drivers.

Implementing an IoCompletion Routine

6/25/2019 • 5 minutes to read • Edit Online

On entry, an *IoCompletion* routine receives a *Context* pointer. When a dispatch routine calls **IoSetCompletionRoutine**, it can supply a *Context* pointer. This pointer can reference whatever driver-determined context information the *IoCompletion* routine requires to process an IRP. Note that the context area cannot be pageable because the *IoCompletion* routine can be called at IRQL = DISPATCH_LEVEL.

Consider the following implementation guidelines for IoCompletion routines:

- An *IoCompletion* routine can check the IRP's I/O status block to determine the result of the I/O operation.
- If the input IRP was allocated by the dispatch routine using **IoAllocateIrp** or **IoBuildAsynchronousFsdRequest**, the *IoCompletion* routine must call **IoFreeIrp** to release that IRP, preferably before it completes the original IRP.
    - The *IoCompletion* routine must release any per-IRP resources the dispatch routine allocated for the driver-allocated IRP, preferably before it frees the corresponding IRP.

      For example, if the dispatch routine allocates an MDL with **IoAllocateMdl** and calls **IoBuildPartialMdl** for a partial-transfer IRP it allocates, the *IoCompletion* routine must release the MDL with **IoFreeMdl**. If it allocates resources to maintain state about the original IRP, it must free those resources, preferably before it calls **IoCompleteRequest** with the original IRP and definitely before it returns control.

      In general, before freeing or completing an IRP, the *IoCompletion* routine should free any per-IRP resources allocated by the Dispatch routine. Otherwise, the driver must maintain state about the resources to be freed before its *IoCompletion* routine returns control from completing the original request.
    - If the *IoCompletion* routine cannot complete the original IRP with STATUS_SUCCESS, it must set the I/O status block in the original IRP to the value returned in the driver-allocated IRP that caused the *IoCompletion* routine to fail the original request.
    - If the *IoCompletion* routine will complete the original request with STATUS_PENDING, it must call **IoMarkIrpPending** with the original IRP before it calls **IoCompleteRequest**.
    - If the *IoCompletion* routine must fail the original IRP with an error STATUS_*XXX*, it can log an error. However, it is the responsibility of the underlying device driver to log any device I/O errors that occur, so *IoCompletion* routines usually do not log errors.
    - When the *IoCompletion* routine has processed and freed the driver-allocated IRP, the routine must return control with STATUS_MORE_PROCESSING_REQUIRED.

      Returning STATUS_MORE_PROCESSING_REQUIRED from the *IoCompletion* routine forestalls the I/O manager's completion processing for a driver-allocated and freed IRP. A second call to **IoCompleteRequest** causes the I/O manager to resume calling the IRP's completion routines, starting with the completion routine immediately above the routine that returned STATUS_MORE_PROCESSING_REQUIRED.
- If the *IoCompletion* routine reuses an incoming IRP to send one or more requests to lower drivers, or if the routine retries failed operations, it should update whatever context the *IoCompletion* routine maintains about each reuse or retry of the IRP. Then it can set up the next-lower driver's I/O stack location again, call **IoSetCompletionRoutine** with its own entry point, and call **IoCallDriver** for the IRP.

- The *IoCompletion* routine should not call **IoMarkIrpPending** at each reuse or retry of the IRP.

  The dispatch routine already marked the original IRP as pending. Until all drivers in the chain complete the original IRP with **IoCompleteRequest**, it remains pending.

- Before retrying a request, the *IoCompletion* routine should reset the I/O status block with STATUS_SUCCESS for **Status** and zero for **Information**, possibly after saving the returned error information.

  For each retry, the *IoCompletion* routine usually decrements a retry count set up by the Dispatch routine. Typically, the *IoCompletion* routine must call **IoCompleteRequest** to fail the IRP when some limited number of retries have failed.

- The *IoCompletion* routine must return STATUS_MORE_PROCESSING_REQUIRED after it calls **IoSetCompletionRoutine** and **IoCallDriver** with an IRP that is being reused or retried.

  Returning STATUS_MORE_PROCESSING_REQUIRED from the *IoCompletion* routine forestalls the I/O manager's completion processing of a reused or retried IRP.

- If the *IoCompletion* routine cannot complete the original IRP with STATUS_SUCCESS, it must leave the I/O status block as returned by lower drivers for the reuse or retry operation that causes the *IoCompletion* routine to fail the IRP.

- If the *IoCompletion* routine will complete the original request with STATUS_PENDING, it must call **IoMarkIrpPending** with the original IRP before it calls **IoCompleteRequest**.

- If the *IoCompletion* routine must fail the original IRP with an error STATUS_*XXX*, it can log an error. However, it is the responsibility of the underlying device driver to log any device I/O errors that occur, so *IoCompletion* routines usually do not log errors.

- Any driver that sets an *IoCompletion* routine in an IRP and then passes the IRP down to a lower driver should check the **IRP->PendingReturned** flag in the *IoCompletion* routine. If the flag is set, the *IoCompletion* routine must call **IoMarkIrpPending** with the IRP. Note, however, that a driver that passes down the IRP and then waits on an event should not mark the IRP pending. Instead, its *IoCompletion* routine should signal the event and return STATUS_MORE_PROCESSING_REQUIRED.

- The *IoCompletion* routine must release any resources the dispatch routine allocated for processing the original IRP, preferably before the *IoCompletion* routine calls **IoCompleteRequest** with the original IRP and definitely before the *IoCompletion* routine returns control from completing the original IRP.

If any higher-level driver has set its *IoCompletion* routine in the original IRP, that driver's *IoCompletion* routine is not called until the *IoCompletion* routines of all lower-level drivers have been called.

Supplying a Priority Boost in Calls to IoCompleteRequest

If a lowest-level device driver can complete an IRP in its dispatch routine, it calls **IoCompleteRequest** with a *PriorityBoost* of IO_NO_INCREMENT. No run-time priority increase is needed because the driver can assume that the original requester did not wait for its I/O operation to be completed.

Otherwise, the lowest-level driver supplies a system-defined and device-type-specific value that boosts the requester's run-time priority to compensate for the time the requester waited on its device I/O request. See Wdm.h or Ntddk.h for the boost values.

Higher-level drivers apply the same *PriorityBoost* as their respective underlying device drivers when they call **IoCompleteRequest**.

Effect of Calling IoCompleteRequest

When a driver calls **IoCompleteRequest**, the I/O manager fills that driver's I/O stack location with zeros before calling the next higher-level driver, if any, that has set up an *IoCompletion* routine to be called for the IRP.

A higher-level driver's *IoCompletion* routine can check only the IRP's I/O status block to determine how all lower drivers handled the request.

The caller of **IoCompleteRequest** must not attempt to access the just-completed IRP. Such an attempt is a programming error that causes a system crash.

Canceling IRPs

6/25/2019 • 2 minutes to read • Edit Online

Drivers in which IRPs might remain queued for an indefinite interval (so a user could cancel a previously submitted I/O request) must have one or more *Cancel* routines to complete user-canceled I/O requests. For example, keyboard, mouse, parallel, serial, and sound device drivers (or drivers layered over them) and file system drivers should have *Cancel* routines.

Drivers for Microsoft Windows XP and later operating systems can use cancel-safe IRP queues rather than implement their own *Cancel* routines.

To "cancel an IRP" means to complete the IRP as quickly as possible while still maintaining system integrity. For a general discussion of IRP completion, see Completing IRPs.

The cancellation process begins when either the system or a driver calls **IoCancelIrp**. This routine is called for each IRP that is associated with the thread that has not yet fully completed. The system cancels unprocessed IRPs if the thread that initiated the I/O request exits. Drivers can cancel only IRPs that they have created (see Creating IRPs for Lower-Level Drivers.)

If an IRP is not completed within 5 minutes, the I/O manager considers the IRP timed out. Such IRPs are disassociated from the thread, and an error is logged for the device that currently owns the IRP. You should ensure that any requests that might take a long time to complete in your driver are cancelable. To ensure that potentially long requests are cancelable, you can use cancel-safe IRP queues or Kernel-Mode Driver Framework, which abstracts cancellation away from the driver developer.

This section provides the following topics:

Introduction to Cancel Routines

6/25/2019 • 2 minutes to read • Edit Online

Any driver in which IRPs can be held in a pending state for an indefinite interval must have one or more *Cancel* routines. For example, a keyboard driver might wait indefinitely for a user to press a key. Conversely, if a driver will never queue more IRPs than it can complete in five minutes, it probably does not need a *Cancel* routine.

Suppose a user-mode thread makes an I/O request, which is queued by a highest-level device driver's dispatch routine, and the requesting thread is terminated while the IRP is queued. IRPs queued on behalf of a terminated thread should be canceled. Consequently, the driver must set a driver-supplied *Cancel* routine in each IRP that it queues.

A driver that creates associated IRPs must cancel them when the master IRP is canceled. Because associated IRPs are not associated with a requesting thread, the master IRP's *Cancel* routine is responsible for canceling any associated IRPs when the master IRP is canceled.

The number of *Cancel* routines any driver has depends on the driver's design. In general, a driver should have a *Cancel* routine for each stage in its I/O processing at which an IRP might be held in a pending state for an indefinite interval. Such pending IRPs are said to be *held in a cancelable state*.

Consider the following design guidelines:

- The highest-level driver in a chain of layered drivers must have at least one *Cancel* routine if it queues IRPs or otherwise holds IRPs in a cancelable state. It can have more than one *Cancel* routine, if necessary.
- Lower-level drivers in which IRPs can be held in a cancelable state for relatively long intervals also should have one or more *Cancel* routines.
- If a driver manages its own internal queues of IRPs, it should have a separate *Cancel* routine for each of its queues.

Some highest-level drivers for interactive devices, such as keyboard, mouse, sound, parallel class and serial drivers, must have *Cancel* routines. Some lower-level drivers, such as a parallel port driver that holds IRPs queued for some number of higher-level class drivers for relatively long intervals, also should have *Cancel* routines.

Mass-storage device drivers, along with intermediate drivers layered over them, are unlikely to have *Cancel* routines. It is the responsibility of a file system driver to handle the cancellation of file I/O requests, while the IRPs input to lower-level mass-storage drivers are usually processed to completion too quickly to be cancelable.

Registering a Cancel Routine

6/25/2019 • 2 minutes to read • Edit Online

If a device driver has a *StartIo* routine, its dispatch routines can register a *Cancel* routine by supplying its address as input to **IoStartPacket**.

If a driver does not have a *StartIo* routine, its dispatch routines must do the following before queuing an IRP for further processing by other driver routines:

1. Call **IoAcquireCancelSpinLock**.
2. Call **IoSetCancelRoutine** with the input IRP and the entry point for a driver-supplied *Cancel* routine.
3. Call **IoReleaseCancelSpinLock**.

For information about the cancel spin lock, see Using the System's Cancel Spin Lock.

Drivers that manage their own queues of IRPs, rather than using the I/O manager-supplied device queue, do not need to acquire the cancel spin lock when calling **IoSetCancelRoutine**. However, these drivers should check the *Cancel* routine pointer that **IoSetCancelRoutine** returns to determine whether the *Cancel* routine has already started.

Points to Consider When Canceling IRPs

6/25/2019 • 3 minutes to read • Edit Online

This section discusses guidelines for implementing a *Cancel* routine and handling cancelable IRPs. For more information about handling cancelable IRPs, see the Flow of Control for Cancel-Safe IRP Queuing.

General Guidelines for All Cancel Routines

The I/O manager holds the cancel spin lock any time it calls a driver's *Cancel* routine. Consequently, every *Cancel* routine must:

- Call **IoReleaseCancelSpinLock** before it returns control.
- Not call **IoAcquireCancelSpinLock** unless it calls **IoReleaseCancelSpinLock** first.
- Make a reciprocal call to **IoReleaseCancelSpinLock** for each call it makes to **IoAcquireCancelSpinLock**.

Each time the *Cancel* routine calls **IoReleaseCancelSpinLock**, it must pass the IRQL returned by the most recent call to **IoAcquireCancelSpinLock**. When releasing the spin lock acquired by the I/O manager (and held when the *Cancel* routine was called), the *Cancel* routine must pass **Irp->CancelIrql**.

A driver must not call outside routines (such as **IoCompleteRequest**) while holding a spin lock because a deadlock can result.

Using the Queue Defined by the I/O Manager

Unless a driver manages its own internal queues of IRPs, its *Cancel* routine is called with an incoming IRP that could be either of the following:

- The **CurrentIrp** in the input target device object
- An entry in the device queue associated with the target device object

Unless a driver manages its own internal queues of IRPs, its *Cancel* routine should call **KeRemoveEntryDeviceQueue** with the input IRP to test whether it is an entry in the device queue associated with the target device object. The driver's *Cancel* routine *cannot* call **KeRemoveDeviceQueue** or **KeRemoveByKeyDeviceQueue** because it cannot assume that the given IRP is at any particular position in the device queue.

Current State of the Input IRP

If a *Cancel* routine is called with an IRP for which the driver has already started I/O processing and the request will be completed soon, the *Cancel* routine should release the system cancel spin lock and return control.

If the current state of the input IRP is Pending, a *Cancel* routine must do the following:

1. Set the input IRP's I/O status block with STATUS_CANCELLED for **Status** and zero for **Information**.
2. Release any spin locks it is holding, including the system cancel spin lock.
3. Call **IoCompleteRequest** with the given IRP.

Holding IRPs in a Cancelable State

Any driver routine that holds an IRP in a cancelable state must call **IoMarkIrpPending** and must call **IoSetCancelRoutine** to set its entry point for the *Cancel* routine in the IRP. Only then can that driver routine call additional support routines such as **IoStartPacket**, **IoAllocateController**, or an **ExInterlockedInsert..List** routine.

Any driver routine that subsequently processes cancelable IRPs must check whether an IRP has already been canceled before it begins operations to satisfy the request. The routine must call **IoSetCancelRoutine** to reset its entry point for the *Cancel* routine to **NULL** in the IRP. Only then can that routine begin its I/O processing for the input IRP.

A routine might have to reset the entry point for a *Cancel* routine in an IRP if it, too, passes IRPs on for further processing by other driver routines and those IRPs might be held in a cancelable state.

Any higher-level driver that holds an IRP in a cancelable state must reset its *Cancel* entry point to **NULL** before it passes the IRP on to the next-lower driver with **IoCallDriver**.

Canceling an IRP

Any higher-level driver can call **IoCancelIrp** with an IRP that it has allocated and passed on for further processing by lower-level drivers. However, such a driver cannot assume that the given IRP will be completed with STATUS_CANCELLED by lower drivers.

Synchronization

A driver can (or must, depending on its design) maintain additional state information in its device extension to track the cancelable status of IRPs. If this state is shared by driver routines running at IRQL <= DISPATCH_LEVEL, the shared data should be protected with a driver-allocated and initialized spin lock.

The driver should manage its acquisitions and releases of the system cancel spin lock and its own spin locks carefully. It should hold the system cancel spin lock for the shortest possible intervals. Before accessing a cancelable IRP, such a driver should always check the return value of **IoSetCancelRoutine** to determine whether the *Cancel* routine is already running (or is about to run); if so, it should let the *Cancel* routine complete the IRP.

If a device driver maintains state information about cancelable IRPs that various driver routines share with its ISR, these other routines must synchronize access to the shared state with the ISR. Only a driver-supplied *SynchCritSection* routine can access state information that is shared with the ISR in a multiprocessor-safe way.

For more information, see Synchronization Techniques.

Synchronizing IRP Cancellation

6/25/2019 • 2 minutes to read • Edit Online

From a driver's perspective, an IRP can be canceled at any time. IRP cancellation occurs asynchronously; therefore, drivers must be able to handle a number of potential race conditions, created if the IRP is canceled at any of the following points:

- After a driver routine is called, but before it queues an IRP.
- After a driver routine is called, but before it tries to process an IRP. For example, an IRP might be canceled after a driver's *StartIo* routine is called, but before the *StartIo* routine removes the IRP from the device queue.
- After the driver routine dequeues the IRP, but before it starts the requested I/O.

Note that after a driver queues an IRP and releases any spin locks that protect the queue, another thread can access and change the IRP. When the original thread resumes—even as soon as the next line of code—the IRP might have already been canceled or otherwise changed.

Driver can use the cancel-safe IRP queue framework to implement IRP queuing. The system then registers a *Cancel* routine for the driver that automatically handles synchronization to safely cancel IRPs. See Cancel-Safe IRP Queues for more information. Otherwise, drivers must implement their own synchronization.

The following members of an IRP contain information about cancellation:

- **Irp->Cancel** indicates whether an IRP is being canceled or should be canceled.
- **Irp->CancelRoutine** indicates whether an IRP is cancelable. If this member contains a pointer to a cancel routine, then the IRP is cancelable. If this member is **NULL**, then the IRP is not cancelable. If this member is **NULL**, but **Irp->Cancel** is set, that indicates that the cancel routine is running and the IRP is in the process of being canceled.

If a driver handles cancelable IRPs, it is responsible for setting the appropriate *Cancel* routine in each IRP that it holds in a cancelable state.

This section includes the following topics on synchronizing IRP cancellation.

Using the System's Cancel Spin Lock

6/25/2019 • 2 minutes to read • Edit Online

The system provides a single *cancel spin lock*, which is acquired or released when certain system routines are called.

Driver routines that change the state of cancelable IRPs, including all routines that might complete an IRP with STATUS_CANCELLED, must acquire and release the system cancel spin lock according to the guidelines in this section.

In drivers that use the I/O manager-supplied device queue, any driver routine other than the *Cancel* routine that changes the cancelable state of an IRP must first call **IoAcquireCancelSpinLock** to acquire the system cancel spin lock.

Acquiring the cancel spin lock ensures that only the caller can change the cancelable state of that IRP. While the caller holds the spin lock, the I/O manager cannot call the driver's *Cancel* routine for that IRP. Likewise, another driver routine, such as a *DispatchCleanup* routine, cannot simultaneously try to change the cancelable state of that IRP.

In drivers that manage their own queues of IRPs and use driver-supplied spin locks to synchronize queue access, the driver routines do not need to acquire the cancel spin lock before calling **IoSetCancelRoutine**. However, these drivers should check the *Cancel* routine pointer that **IoSetCancelRoutine** returns to determine whether the *Cancel* routine has already started. See Using a Driver-Supplied Spin Lock for details.

Any driver routine that calls **IoAcquireCancelSpinLock** must call **IoReleaseCancelSpinLock** as soon as possible.

A driver must never call **IoCompleteRequest** with an IRP while holding a spin lock. Attempting to complete an IRP while holding a spin lock can cause a deadlock.

Synchronizing Cancellation in Driver Routines that Process IRPs

6/25/2019 • 2 minutes to read • Edit Online

Any driver routine that dequeues or is called with an IRP that is held in a cancelable state, including a driver's *StartIo* routine, must do the following:

1. Call **IoAcquireCancelSpinLock**.
2. Check to make sure that **Irp** equals **DeviceObject->CurrentIrp**. If not, call **IoReleaseCancelSpinLock** and return control.

   If the two are not the same, the **CurrentIrp** might have been canceled between the time that **IoStartPacket** released the cancel spin lock and this routine acquired it.
3. Call **IoSetCancelRoutine** with a **NULL** *CancelRoutine* pointer to remove the IRP from the cancelable state.
4. Check the **Irp->Cancel** field to determine whether to cancel the IRP or to begin processing the I/O request.

   If **Irp->Cancel** is set to **TRUE**, do the following:

   - Call **IoReleaseCancelSpinLock**.
   - Set **Irp->IoStatus.Status** to STATUS_CANCELLED.
   - Set **Irp->IoStatus.Information** to 0.
   - Call **IoStartNextPacket** (in a *StartIo* routine) to start the next packet.
   - Call **IoCompleteRequest** with a priority boost of IO_NO_INCREMENT to complete the IRP.

   If **Irp->Cancel** is set to **FALSE**, call **IoReleaseCancelSpinLock** and start the requested processing the I/O request or pass the IRP to the next lower driver, as appropriate.

Drivers that manage their own queues of IRPs, rather than using the I/O manager-supplied device queue, do not need to acquire the cancel spin lock when calling **IoSetCancelRoutine**. However, these drivers should check the *Cancel* routine pointer that **IoSetCancelRoutine** returns to determine whether the cancel routine has already started.

In any driver that handles cancelable IRPs, every driver routine that processes an IRP before the underlying device has been programmed for the requested I/O operation should check the cancelable state of all incoming IRPs. Specifically, a highest-level device driver with both *StartIo* and *ControllerControl* routines should process incoming IRPs in both these driver routines as already described.

Synchronizing Cancellation in Higher-Level Drivers without Cancel Routines

6/25/2019 • 2 minutes to read • Edit Online

A higher-level driver can make no assumptions about whether or how existing lower-level drivers handle cancelable IRPs. As soon as any higher-level driver calls **IoCallDriver** for an IRP, it no longer owns that IRP and it can neither ascertain nor control processing of the IRP by lower-level drivers.

However, any higher-level driver can set an *IoCompletion* routine for an IRP by calling **IoSetCompletionRoutine** before it calls **IoCallDriver**. The higher-level driver can determine whether any pending IRP is canceled in a lower driver by calling **IoSetCompletionRoutine** with the *InvokeOnCancel* parameter set to **TRUE** before it passes the IRP on to lower drivers. Doing so ensures that the driver's *IoCompletion* routine will be called whether the IRP is canceled or completed.

A higher-level driver can call **IoCancelIrp** with any pending IRP that the driver has allocated. However, making this call does not ensure that the driver-allocated IRP will be completed with its I/O status block set to STATUS_CANCELLED; another thread might already be completing the IRP. To check whether the IRP was canceled, the higher-level driver must call **IoSetCompletionRoutine** with the *InvokeOnCancel* parameter set to **TRUE** before passing the IRP on to the next lower driver. See Completing IRPs for more information about completion routines.

A higher-level driver must not call **IoCancelIrp** with an IRP that it did not allocate.

Using a Driver-Supplied Spin Lock

6/25/2019 • 6 minutes to read • Edit Online

Drivers that manage their own queues of IRPs can use a driver-supplied spin lock, instead of the system cancel spin lock, to synchronize access to the queues. You can improve performance by avoiding use of the cancel spin lock except when absolutely necessary. Because the system has only one cancel spin lock, a driver might sometimes have to wait for that spin lock to become available. Using a driver-supplied spin lock eliminates this potential delay and makes the cancel spin lock available for the I/O manager and other drivers. Although the system still acquires the cancel spin lock when it calls the driver's *Cancel* routine, a driver can use its own spin lock to protect its queue of IRPs.

Even if a driver does not queue pending IRPs, but retains ownership in some other way, that driver must set a *Cancel* routine for the IRP and must use a spin lock to protect the IRP pointer. For example, suppose a driver marks an IRP pending, then passes the IRP pointer as context to an *IoTimer* routine. The driver must set a *Cancel* routine that cancels the timer and must use the same spin lock in both the *Cancel* routine and the timer callback when accessing the IRP.

Any driver that queues its own IRPs and uses its own spin lock must do the following:

- Create a spin lock to protect the queue.
- Set and clear the *Cancel* routine only while holding this spin lock.
- If the *Cancel* routine starts running while the driver is dequeuing an IRP, allow the *Cancel* routine to complete the IRP.
- Acquire the lock that protects the queue in the *Cancel* routine.

To create the spin lock, the driver calls **KeInitializeSpinLock**. In the following example, the driver saves the spin lock in a **DEVICE_CONTEXT** structure along with the queue it has created:

```
typedef struct {
    LIST_ENTRYirpQueue;
    KSPIN_LOCK irpQueueSpinLock;
    ...
} DEVICE_CONTEXT;

VOID InitDeviceContext(DEVICE_CONTEXT *deviceContext)
{
    InitializeListHead(&deviceContext->irpQueue);
    KeInitializeSpinLock(&deviceContext->irpQueueSpinLock);
}
```

To queue an IRP, the driver acquires the spin lock, calls **InsertTailList**, and then marks the IRP pending, as in the following example:

```
NTSTATUS QueueIrp(DEVICE_CONTEXT *deviceContext, PIRP Irp)
{
   PDRIVER_CANCEL  oldCancelRoutine;
   KIRQL  oldIrql;
   NTSTATUS  status;

   KeAcquireSpinLock(&deviceContext->irpQueueSpinLock, &oldIrql);

   // Queue the IRP and call IoMarkIrpPending to indicate
   // that the IRP may complete on a different thread.
   // N.B. It is okay to call these inside the spin lock
   // because they are macros, not functions.
   IoMarkIrpPending(Irp);
   InsertTailList(&deviceContext->irpQueue, &Irp->Tail.Overlay.ListEntry);

   // Must set a Cancel routine before checking the Cancel flag.
   oldCancelRoutine = IoSetCancelRoutine(Irp, IrpCancelRoutine);
   ASSERT(oldCancelRoutine == NULL);

   if (Irp->Cancel) {
      // The IRP was canceled. Check whether our cancel routine was called.
      oldCancelRoutine = IoSetCancelRoutine(Irp, NULL);
      if (oldCancelRoutine) {
         // The cancel routine was NOT called.
         // So dequeue the IRP now and complete it after releasing the spin lock.
         RemoveEntryList(&Irp->Tail.Overlay.ListEntry);
         // Drop the lock before completing the request.
         KeReleaseSpinLock(&deviceContext->irpQueueSpinLock, oldIrql);
         Irp->IoStatus.Status = STATUS_CANCELLED;
         Irp->IoStatus.Information = 0;
         IoCompleteRequest(Irp, IO_NO_INCREMENT);
         return STATUS_PENDING;

      } else {
         // The Cancel routine WAS called.
         // As soon as we drop our spin lock, it will dequeue and complete the IRP.
         // So leave the IRP in the queue and otherwise do not touch it.
         // Return pending since we are not completing the IRP here.

      }
   }

   KeReleaseSpinLock(&deviceContext->irpQueueSpinLock, oldIrql);

   // Because the driver called IoMarkIrpPending while it held the IRP,
   // it must return STATUS_PENDING from its dispatch routine.
   return STATUS_PENDING;
}
```

As the example shows, the driver holds its spin lock while it sets and clears the *Cancel* routine. The sample queuing routine contains two calls to **IoSetCancelRoutine**.

The first call sets the *Cancel* routine for the IRP. However, because the IRP might have been canceled while the queuing routine is running, the driver must check the **Cancel** member of the IRP.

- If **Cancel** is set, then cancellation has been requested, and the driver must make a second call to **IoSetCancelRoutine** to see whether the previously set *Cancel* routine was called.
- If the IRP has been canceled but the *Cancel* routine has not yet been called, then the current routine dequeues the IRP and completes it with STATUS_CANCELLED.
- If the IRP has been canceled and the *Cancel* routine has already been called, then the current return marks the IRP pending and returns STATUS_PENDING. The *Cancel* routine will complete the IRP.

The following example shows how to remove an IRP from the previously created queue:

```
PIRP DequeueIrp(DEVICE_CONTEXT *deviceContext)
{
   KIRQL oldIrql;
   PIRP nextIrp = NULL;

   KeAcquireSpinLock(&deviceContext->irpQueueSpinLock, &oldIrql);

   while (!nextIrp && !IsListEmpty(&deviceContext->irpQueue)) {
      PDRIVER_CANCEL oldCancelRoutine;
      PLIST_ENTRY listEntry = RemoveHeadList(&deviceContext->irpQueue);

      // Get the next IRP off the queue.
      nextIrp = CONTAINING_RECORD(listEntry, IRP, Tail.Overlay.ListEntry);

      // Clear the IRP's cancel routine.
      oldCancelRoutine = IoSetCancelRoutine(nextIrp, NULL);

      // IoCancelIrp() could have just been called on this IRP. What interests us
      // is not whether IoCancelIrp() was called (nextIrp->Cancel flag set), but
      // whether IoCancelIrp() called (or is about to call) our Cancel routine.
      // For that, check the result of the test-and-set macro IoSetCancelRoutine.
      if (oldCancelRoutine) {
         // Cancel routine not called for this IRP. Return this IRP.
         ASSERT(oldCancelRoutine == IrpCancelRoutine);
      } else {
         // This IRP was just canceled and the cancel routine was (or will be)
         // called. The Cancel routine will complete this IRP as soon as we
         // drop the spin lock, so do not do anything with the IRP.
         // Also, the Cancel routine will try to dequeue the IRP, so make
         // the IRP's ListEntry point to itself.
         ASSERT(nextIrp->Cancel);
         InitializeListHead(&nextIrp->Tail.Overlay.ListEntry);
         nextIrp = NULL;
      }
   }

   KeReleaseSpinLock(&deviceContext->irpQueueSpinLock, oldIrql);

   return nextIrp;
}
```

In the example, the driver acquires the associated spin lock before it accesses the queue. While holding the spin lock, it checks that the queue is not empty and gets the next IRP off the queue. Then it calls **IoSetCancelRoutine** to reset the *Cancel* routine for the IRP. Because the IRP could be canceled while the driver dequeues the IRP and resets the *Cancel* routine, the driver must check the value returned by **IoSetCancelRoutine**. If **IoSetCancelRoutine** returns **NULL**, which indicates that the *Cancel* routine either has been or will soon be called, then the dequeuing routine lets the *Cancel* routine complete the IRP. It then releases the lock that protects the queue and returns.

Note the use of **InitializeListHead** in the preceding routine. The driver could requeue the IRP, so that the *Cancel* routine can dequeue it, but it is simpler to call **InitializeListHead**, which reinitializes the IRP's **ListEntry** field so that it points to the IRP itself. Using the self-referencing pointer is important because the structure of the list could change before the *Cancel* routine acquires the spin lock. And if the list structure changes, possibly making the original value of **ListEntry** invalid, the *Cancel* routine could corrupt the list when it dequeues the IRP. But if **ListEntry** points to the IRP itself, then the *Cancel* routine will always use the correct IRP.

The *Cancel* routine, in turn, simply does the following:

```
VOID IrpCancelRoutine(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
   DEVICE_CONTEXT  *deviceContext = DeviceObject->DeviceExtension;
   KIRQL  oldIrql;

   // Release the global cancel spin lock.
   // Do this while not holding any other spin locks so that we exit at the right IRQL.
   IoReleaseCancelSpinLock(Irp->CancelIrql);

   // Dequeue and complete the IRP.
   // The enqueue and dequeue functions synchronize properly so that if this cancel routine is called,
   // the dequeue is safe and only the cancel routine will complete the IRP. Hold the spin lock for the IRP
   // queue while we do this.

   KeAcquireSpinLock(&deviceContext->irpQueueSpinLock, &oldIrql);

   RemoveEntryList(&Irp->Tail.Overlay.ListEntry);

   KeReleaseSpinLock(&deviceContext->irpQueueSpinLock, oldIrql);

   // Complete the IRP. This is a call outside the driver, so all spin locks must be released by this point.
   Irp->IoStatus.Status = STATUS_CANCELLED;
   IoCompleteRequest(Irp, IO_NO_INCREMENT);
   return;
}
```

The I/O manager always acquires the global cancel spin lock before it calls a *Cancel* routine, so the first task of the *Cancel* routine is to release this spin lock. It then acquires the spin lock that protects the driver's queue of IRPs, removes the current IRP from the queue, releases its spin lock, completes the IRP with STATUS_CANCELLED and no priority boost, and returns.

For more information about canceling spin locks, see the Cancel Logic in Windows Drivers white paper.

Implementing a Cancel Routine

6/25/2019 • 2 minutes to read • Edit Online

The I/O manager calls a driver-supplied *Cancel* routine with an input IRP to be canceled and a *DeviceObject* pointer that represents the target device for the I/O request.

The IRP could be one that the driver's *DispatchReadWrite* routine has queued just as the current Win32 application is being closed by the user. The IRP also could be one that a higher-level driver explicitly canceled, depending on the nature of the underlying device.

When the *Cancel* routine is called, the input IRP might already be the **CurrentIrp** in the target device object or might already be in the device queue associated with the target device object if the driver has a *StartIo* routine. If the driver has no *StartIo* routine, the IRP might be in a driver-managed internal queue of IRPs when its *Cancel* routine is called. In any case, before the I/O manager calls the *Cancel* routine for the incoming IRP, the I/O manager sets the **Cancel** member in this IRP to **TRUE** and sets the **CancelRoutine** member in the IRP to **NULL**.

The *Cancel* routine for a master IRP that has associated IRPs is responsible for calling **IoCancelIrp** to cancel those associated IRPs.

All *Cancel* routines must follow these guidelines:

- Call **IoReleaseCancelSpinLock** to release the system's cancel spin lock.
- Set the I/O status block's **Status** member to STATUS_CANCELLED, and set its **Information** member to zero.
- Complete the specified IRP by calling **IoCompleteRequest**.
- Because a *Cancel* routine is always called with the system cancel spin lock held, this routine must not call **IoAcquireCancelSpinLock** unless it calls **IoReleaseCancelSpinLock** first.
- A *Cancel* routine cannot be holding the system cancel spin lock when it returns control. That is, every *Cancel* routine must call **IoReleaseCancelSpinLock** at least once before it returns control.
- If it calls **IoAcquireCancelSpinLock**, a *Cancel* routine must make the reciprocal call to **IoReleaseCancelSpinLock** as quickly as possible.
- Never call **IoCompleteRequest** with an IRP while holding a spin lock. Attempting to complete an IRP while holding a spin lock can cause deadlocks.

Cancel Routines in Drivers with StartIo Routines

6/25/2019 • 2 minutes to read • Edit Online

The I/O manager maintains the **CurrentIrp** field in a device object only if IRPs are queued in the associated device queue object. That is, the field is valid only if the driver has a *StartIo* routine.

In a driver that has a *StartIo* routine, a typical *Cancel* routine must do the following:

1. Check whether the pointer for the input IRP matches the target device object's **CurrentIrp** address.

   If these pointers are equivalent, the *Cancel* routine calls **IoReleaseCancelSpinLock**, passing **Irp->CancelIrql**, and returns control.

2. If the canceled IRP is not the current IRP, check whether the input canceled IRP is in the device queue associated with the target device object by calling **KeRemoveEntryDeviceQueue** with the IRP's **Tail.Overlay.DeviceQueueEntry** pointer.

   - If the IRP is in the device queue, calling **KeRemoveEntryDeviceQueue** removes it from the queue. The *Cancel* routine calls **IoReleaseCancelSpinLock**, sets the IRP's I/O status block with STATUS_CANCELLED for **Status** and zero for **Information**, calls **IoCompleteRequest** with the canceled IRP, and returns control.
   - If the IRP is not in the device queue, the *Cancel* routine calls **IoReleaseCancelSpinLock** and returns control.

The driver's *Cancel* routine should call **KeRemoveEntryDeviceQueue** to test whether the IRP is in the device queue. This support routine either removes the given IRP from the device queue or does nothing except return **FALSE**, indicating that the given entry was not queued. A *Cancel* routine cannot assume that the input IRP is at any particular position in the device queue, so it cannot call **KeRemoveDeviceQueue** or **KeRemoveByKeyDeviceQueue** to compare the pointers to the returned IRP and input IRP.

Drivers with *Cancel* routines can handle **IRP_MJ_CLEANUP** requests as well. See *DispatchCleanup* for more information about **IRP_MJ_CLEANUP** requests.

Cancel Routines in Drivers without StartIo Routines

6/25/2019 • 2 minutes to read • Edit Online

The I/O manager maintains the **CurrentIrp** field in a device object only if IRPs are queued in the associated device queue object.

Drivers that do not have *StartIo* routines manage their own internal queues of IRPs. In such a driver, a *Cancel* routine can be called with an input IRP that is neither the **CurrentIrp** for the input target device object, nor an IRP in the driver's internal queue. The driver must maintain its own state about which IRP is currently being processed and should have a *Cancel* routine for each of its queues. The driver's internal queue should be an interlocked queue because its internal queue must be protected by an executive spin lock.

When the driver's *Cancel* routine is called, it typically does the following:

1. Calls **IoReleaseCancelSpinLock**, passing **Irp->CancelIrql**.
2. Acquires the spin lock that protects its interlocked queue and walks the queue to find an IRP with **Irp->Cancel** set to **TRUE**.

   - If it finds such an IRP in the interlocked queue, dequeues it, releases the spin lock protecting the queue, sets the IRP's I/O status block with w

     STATUS_CANCELLED for **Status** and zero for **Information**, starts the next queued IRP, calls **IoCompleteRequest** with the canceled IRP, and returns control

   - If it does not find such an IRP, the *Cancel* routine releases any spin locks it is holding and returns control.

     The driver usually assumes that I/O processing for the input IRP has already begun if the IRP is not queued.

Drivers with *Cancel* routines can handle **IRP_MJ_CLEANUP** requests as well. See *DispatchCleanup* for more information about **IRP_MJ_CLEANUP** requests.

Reusing IRPs

6/25/2019 • 2 minutes to read • Edit Online

Under certain circumstances, drivers can *reuse* IRPs. The driver can allocate a pool of memory buffers that it uses to hold IRPs as they need to be created.

Drivers must not attempt to reuse IRPs issued by the I/O manager. In particular, drivers should not attempt to reuse IRPs created by the **IoMakeAssociatedIrp**, **IoBuildSynchronousFsdRequest**, **IoBuildAsynchronousFsdRequest**, or **IoBuildDeviceIoControlRequest** routines.

Drivers can safely reuse IRPs that they have created, as follows:

1. If a driver allocates an IRP as raw memory (for example, by calling **ExAllocatePoolWithTag**), and then initializes it with **IoInitializeIrp**, then it can safely call **IoInitializeIrp** or **IoReuseIrp** to reinitialize the memory buffer.
2. On Microsoft Windows 2000 and later operating systems, drivers that create an IRP with **IoAllocateIrp** can reuse the IRP by calling **IoReuseIrp**.
3. If a driver allocates an IRP by calling **IoAllocateIrp** with the *ChargeQuota* parameter set to **FALSE**, then it can safely use **IoInitializeIrp** to reinitialize the IRP. Drivers that must work on Windows 98/Me can use this method as a work-around. Drivers designed strictly for Windows 2000 and later operating systems should use the previous method.

Device Type-Specific I/O Requests

6/25/2019 • 2 minutes to read • Edit Online

Device-specific sections of the Windows Driver Kit (WDK) provide information about device type-specific I/O requests handled by the system-supplied drivers for the most common kinds of devices.

A new kernel-mode driver must handle the same set of I/O requests as a system-supplied driver if the new driver meets any of the following conditions:

- The new driver replaces a system driver for the same type of device.
- The new driver supports another device of a type already in the system.
- The new driver is an intermediate (filter) driver, layered between two system-supplied drivers.

Such a new driver must handle every **IRP_MJ_*XXX*** request that the system-supplied drivers handle. In most cases, a new device driver should also handle the same set of **IOCTL_*XXX*** codes for **IRP_MJ_DEVICE_CONTROL** requests, even if the new driver must emulate the behavior of the corresponding system-supplied driver. Otherwise, the new driver might break user-mode applications that expect these kinds of requests to be honored.

For information about the NTSTATUS values that drivers can set in the I/O status block of IRPs, as the return value for specific requests, see Using NTSTATUS Values. For information about NTSTATUS values that can be specified in an error log packet, see Logging Errors. Use this information to decide on the appropriate status values to be returned by new drivers for similar types of devices, or as an aid in determining the appropriate status values to be returned by the driver for a new type of device.

For more information about various kinds of drivers and the requests that each is required to support, see the following:

Serial Devices and Drivers

System-Supplied Parallel Drivers

Storage Drivers

HID Architecture

I/O Requests for USB Client Drivers

The IEEE 1394 Driver Stack

Access Attribute Memory of a PCMCIA Device

For all other types of drivers, consult the documentation for the appropriate driver type.

Introduction to I/O Control Codes

6/25/2019 • 2 minutes to read • Edit Online

I/O control codes (IOCTLs) are used for communication between user-mode applications and drivers, or for communication internally among drivers in a stack. I/O control codes are sent using IRPs.

User-mode applications send IOCTLs to drivers by calling **DeviceIoControl**, which is described in Microsoft Windows SDK documentation. Calls to **DeviceIoControl** cause the I/O manager to create an **IRP_MJ_DEVICE_CONTROL** request and send it to the topmost driver.

Additionally, upper-level drivers can send IOCTLs to lower-level drivers by creating and sending **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests. Drivers process these requests in *DispatchDeviceControl* and *DispatchInternalDeviceControl* routines. (User-mode applications cannot send **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests.)

Some IOCTLs are "public" and some are "private". Public IOCTLs are typically system-defined and documented by Microsoft, in either the Windows Driver Kit (WDK) or the Windows SDK. They might be sent by means of a user-mode component's calls to **DeviceIoControl**, or they might be sent from one kernel-mode driver to another, using **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests. Examples of public IOCTLs include SCSI Port I/O Control Codes and I8042prt Mouse Internal Device Control Requests.

Private IOCTLs, on the other hand, are meant to be used exclusively by a vendor's software components to communicate with each other. Private IOCTLs are typically defined in a vendor-supplied header file and are not publicly documented. Like public IOCTLs, they might be sent by means of a user-mode component's calls to **DeviceIoControl**, or they might be sent from one kernel-mode driver to another, using **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests.

There is no difference between the coding of public and private IOCTLs. There are, however, differences in the internal codes that can be used in vendor-defined IOCTLs, compared with those that are used for system-defined IOCTLs. If the available public IOCTLs do not fit your needs, you can define new private IOCTLs that your software components can use to communicate with one another. For more information, see Defining I/O Control Codes.

Creating IOCTL Requests in Drivers

6/25/2019 • 2 minutes to read • Edit Online

A class driver or other higher-level driver can allocate IRPs for I/O control requests and send them to the next-lower driver as follows:

1. Allocate or reuse an I/O request packet (**IRP**) with the major function code **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL**. You can use the **IoBuildDeviceIoControlRequest** routine to specifically allocate an IOCTL IRP. You can also use general-purpose IRP creation and initialization routines, such as **IoAllocateIrp**, **IoReuseIrp**, or **IoInitializeIrp**. For more information about IRP allocation, see Creating IRPs for Lower-Level Drivers.

2. Set up the lower driver's I/O stack location for the IRP with the IOCTL_*XXX* code and appropriate parameters.

3. If the IOCTL request is to be completed asynchronously, call the **KeInitializeEvent** routine to initialize an event object as a notification event. The driver uses this event to wait for an I/O operation to be completed.

4. Call **IoSetCompletionRoutine** with the IRP so that the upper driver can supply an *IoCompletion* routine, if necessary, to do the following:

   - Determine how the lower driver handled a given request.
   - Reuse the IRP to send another request or dispose of the driver-created IRP, after the lower driver completes a requested operation. The driver cannot reuse IRPs that **IoBuildDeviceIoControlRequest** created. For more information, see Reusing IRPs.

5. Call **IoCallDriver** to pass the request on to the lower driver.

6. If **IoCallDriver** returns STATUS_PENDING, call the **KeWaitForSingleObject** routine to put the current thread into a wait state. The driver sets the routine's *Object* parameter to the address of the event object that was initialized in the call to **KeInitializeEvent**.

   Note If the driver calls **KeWaitForSingleObject** with its *Timeout* parameter set either to **NULL** or to the address of a variable that contains a nonzero value, the driver must be running at IRQL <= APC_LEVEL in a nonarbitrary thread context. Otherwise, the driver must be running at IRQL <= DISPATCH_LEVEL.

The event is signaled by its *IoCompletion* routine when the IOCTL request has completed. Once the event is signaled, the thread resumes execution.

Important If the driver allocates the event object as a local variable on the stack, the driver must call **KeWaitForSingleObject** with its *WaitMode* parameter set to **KernelMode**. This parameter value prevents the stack from being paged out.

To avoid synchronization problems and possible access violations, parameters for I/O control codes rarely include embedded pointers. Except for certain SCSI requests, the buffers at *Irp*->**AssociatedIrp**.**SystemBuffer**, at *Irp*->**MdlAddress**, and at **Parameters**.**DeviceIoControl**.**Type3InputBuffer** in a driver's I/O stack location do not contain pointers to other data buffers, nor do they contain structures that contain pointers for system-defined I/O control codes. For more information about how data buffers are used with IRPs that contain I/O control codes, see Buffer Descriptions for I/O Control Codes.

Nevertheless, a pair of class/port drivers that define internal I/O control codes can pass an embedded pointer to driver-allocated memory from the higher-level driver to the lower-level driver. Such a pair of class/port drivers is responsible for ensuring that the following is true:

- Only one driver at a time can access the data.
- Private data buffers are accessible in an arbitrary thread context by the port driver.

Display drivers can call the GDI function **EngDeviceIoControl** to send privately defined, device-specific I/O control requests, as well as system-defined public I/O control requests, through the system video port driver down to the corresponding adapter-specific video miniport drivers.

Any user-mode component of a driver package can call **DeviceIoControl** to send I/O control requests to a driver stack. The I/O manager creates an **IRP_MJ_DEVICE_CONTROL** request and delivers it to the highest-level driver.

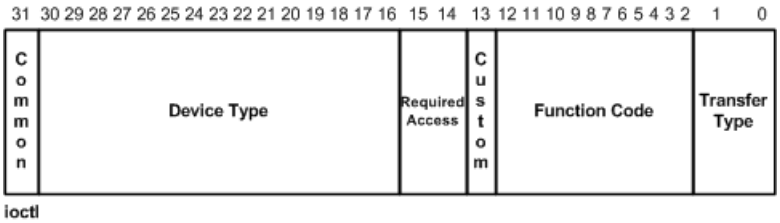
Defining I/O Control Codes

6/25/2019 • 5 minutes to read • Edit Online

When defining new IOCTLs, it is important to remember the following rules:

- If a new IOCTL will be available to user-mode software components, the IOCTL must be used with **IRP_MJ_DEVICE_CONTROL** requests. User-mode components send **IRP_MJ_DEVICE_CONTROL** requests by calling the **DeviceIoControl**, which is a Win32 function.
- If a new IOCTL will be available only to kernel-mode driver components, the IOCTL must be used with **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests. Kernel-mode components create **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests by calling **IoBuildDeviceIoControlRequest**. For more information, see Creating IOCTL Requests in Drivers.

An I/O control code is a 32-bit value that consists of several fields. The following figure illustrates the layout of I/O control codes.



Use the system-supplied **CTL_CODE** macro, which is defined in Wdm.h and Ntddk.h, to define new I/O control codes. The definition of a new IOCTL code, whether intended for use with **IRP_MJ_DEVICE_CONTROL** or **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests, uses the following format:

```
#define IOCTL_Device_Function CTL_CODE(DeviceType, Function, Method, Access)
```

Choose a descriptive constant name for the IOCTL, of the form IOCTL_*Device_Function*, where *Device* indicates the type of device and *Function* indicates the operation. An example constant name is IOCTL_VIDEO_ENABLE_CURSOR.

Supply the following parameters to the **CTL_CODE** macro:

DeviceType
Identifies the device type. This value must match the value that is set in the **DeviceType** member of the driver's **DEVICE_OBJECT** structure. (See Specifying Device Types). Values of less than 0x8000 are reserved for Microsoft. Values of 0x8000 and higher can be used by vendors. Note that the vendor-assigned values set the **Common** bit.

FunctionCode
Identifies the function to be performed by the driver. Values of less than 0x800 are reserved for Microsoft. Values of 0x800 and higher can be used by vendors. Note that the vendor-assigned values set the **Custom** bit.

TransferType
Indicates how the system will pass data between the caller of **DeviceIoControl** (or **IoBuildDeviceIoControlRequest**) and the driver that handles the IRP.

Use one of the following system-defined constants:

METHOD_BUFFERED

Specifies the buffered I/O method, which is typically used for transferring small amounts of data per request. Most I/O control codes for device and intermediate drivers use this *TransferType* value.

For information about how the system specifies data buffers for METHOD_BUFFERED I/O control codes, see Buffer Descriptions for I/O Control Codes.

For more information about buffered I/O, see Using Buffered I/O.

METHOD_IN_DIRECT or METHOD_OUT_DIRECT
Specifies the direct I/O method, which is typically used for reading or writing large amounts of data, using DMA or PIO, that must be transferred quickly.

Specify METHOD_IN_DIRECT if the caller of **DeviceIoControl** or **IoBuildDeviceIoControlRequest** will pass data to the driver.

Specify METHOD_OUT_DIRECT if the caller of **DeviceIoControl** or **IoBuildDeviceIoControlRequest** will receive data from the driver.

For information about how the system specifies data buffers for METHOD_IN_DIRECT and METHOD_OUT_DIRECT I/O control codes, see Buffer Descriptions for I/O Control Codes.

For more information about direct I/O, see Using Direct I/O.

METHOD_NEITHER
Specifies neither buffered nor direct I/O. The I/O manager does not provide any system buffers or MDLs. The IRP supplies the user-mode virtual addresses of the input and output buffers that were specified to **DeviceIoControl** or **IoBuildDeviceIoControlRequest**, without validating or mapping them.

For information about how the system specifies data buffers for METHOD_NEITHER I/O control codes, see Buffer Descriptions for I/O Control Codes.

This method can be used only if the driver can be guaranteed to be running in the context of the thread that originated the I/O control request. Only a highest-level kernel-mode driver is guaranteed to meet this condition, so METHOD_NEITHER is seldom used for the I/O control codes that are passed to low-level device drivers.

With this method, the highest-level driver must determine whether to set up buffered or direct access to user data on receipt of the request, possibly must lock down the user buffer, and must wrap its access to the user buffer in a structured exception handler (see Handling Exceptions). Otherwise, the originating user-mode caller might change the buffered data before the driver can use it, or the caller could be swapped out just as the driver is accessing the user buffer.

For more information, see Using Neither Buffered Nor Direct I/O.

RequiredAccess
Indicates the type of access that a caller must request when opening the file object that represents the device (see **IRP_MJ_CREATE**). The I/O manager will create IRPs and call the driver with a particular I/O control code only if the caller has requested the specified access rights. *RequiredAccess* is specified by using the following system-defined constants:

FILE_ANY_ACCESS
The I/O manager sends the IRP for any caller that has a handle to the file object that represents the target device object.

FILE_READ_DATA
The I/O manager sends the IRP only for a caller with read access rights, allowing the underlying device driver to transfer data from the device to system memory.

FILE_WRITE_DATA
The I/O manager sends the IRP only for a caller with write access rights, allowing the underlying device driver to

transfer data from system memory to its device.

FILE_READ_DATA and FILE_WRITE_DATA can be ORed together if the caller must have both read and write access rights.

Some system-defined I/O control codes have a *RequiredAccess* value of FILE_ANY_ACCESS, which allows the caller to send the particular IOCTL regardless of the access granted to the device. Examples include I/O control codes that are sent to drivers of *exclusive devices*.

Other system-defined I/O control codes require the caller to have read access rights, write access rights, or both. For example, the following definition of the public I/O control code IOCTL_DISK_SET_PARTITION_INFO shows that this I/O request can be sent to a driver only if the caller has both read and write access rights:

```
#define IOCTL_DISK_SET_PARTITION_INFO\
        CTL_CODE(IOCTL_DISK_BASE, 0x008, METHOD_BUFFERED,\
        FILE_READ_DATA | FILE_WRITE_DATA)
```

Note Before specifying FILE_ANY_ACCESS for a new IOCTL code, you must be absolutely certain that allowing unrestricted access to your device does not create a possible path for malicious users to compromise the system.

Drivers can use **IoValidateDeviceIoControlAccess** to perform stricter access checking than that provided by an IOCTL's *RequiredAccess* bits.

Other useful macros

The following macros are useful for extracting the 16-bit *DeviceType* and 2-bit *TransferType* fields from an IOCTL code:

```
#define DEVICE_TYPE_FROM_CTL_CODE(ctrlCode)   (((ULONG)(ctrlCode & 0xffff0000)) >> 16)
#define METHOD_FROM_CTL_CODE(ctrlCode)        ((ULONG)(ctrlCode & 3))
```

These macros are defined in Wdm.h and Ntddk.h.

Buffer Descriptions for I/O Control Codes

6/25/2019 • 2 minutes to read • Edit Online

I/O control codes are contained in **IRP_MJ_DEVICE_CONTROL** and **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests. The I/O manager creates these requests as a result of calls to **DeviceIoControl** (described in the Microsoft Windows SDK documentation) and **IoBuildDeviceIoControlRequest**.

Because **DeviceIoControl** and **IoBuildDeviceIoControlRequest** accept both an input buffer and an output buffer as arguments, all **IRP_MJ_DEVICE_CONTROL** and **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests supply both an input buffer and an output buffer. The way the system describes these buffers is dependent on the data transfer type. The transfer type is specified by the *TransferType* value in the **CTL_CODE** macro that creates IOCTL code values.

The system describes buffers for each *TransferType* value as follows:

METHOD_BUFFERED
For this transfer type, IRPs supply a pointer to a buffer at **Irp->AssociatedIrp.SystemBuffer**. This buffer represents both the input buffer and the output buffer that are specified in calls to **DeviceIoControl** and **IoBuildDeviceIoControlRequest**. The driver transfers data out of, and then into, this buffer.

For input data, the buffer size is specified by **Parameters.DeviceIoControl.InputBufferLength** in the driver's **IO_STACK_LOCATION** structure. For output data, the buffer size is specified by **Parameters.DeviceIoControl.OutputBufferLength** in the driver's **IO_STACK_LOCATION** structure.

The size of the space that the system allocates for the single input/output buffer is the larger of the two length values.

METHOD_IN_DIRECT or METHOD_OUT_DIRECT
For these transfer types, IRPs supply a pointer to a buffer at **Irp->AssociatedIrp.SystemBuffer**. This represents the input buffer that is specified in calls to **DeviceIoControl** and **IoBuildDeviceIoControlRequest**. The buffer size is specified by **Parameters.DeviceIoControl.InputBufferLength** in the driver's **IO_STACK_LOCATION** structure.

For these transfer types, IRPs also supply a pointer to an MDL at **Irp->MdlAddress**. This represents the output buffer that is specified in calls to **DeviceIoControl** and **IoBuildDeviceIoControlRequest**. However, this buffer can actually be used as either an input buffer or an output buffer, as follows:

- METHOD_IN_DIRECT is specified if the driver that handles the IRP receives data in the buffer when it is called. The MDL describes an input buffer, and specifying METHOD_IN_DIRECT ensures that the executing thread has read-access to the buffer.
- METHOD_OUT_DIRECT is specified if the driver that handles the IRP will write data into the buffer before completing the IRP. The MDL describes an output buffer, and specifying METHOD_OUT_DIRECT ensures that the executing thread has write-access to the buffer.

For both of these transfer types, **Parameters.DeviceIoControl.OutputBufferLength** specifies the size of the buffer that is described by the MDL.

METHOD_NEITHER
The I/O manager does not provide any system buffers or MDLs. The IRP supplies the user-mode virtual addresses of the input and output buffers that were specified to **DeviceIoControl** or **IoBuildDeviceIoControlRequest**, without validating or mapping them.

The input buffer's address is supplied by **Parameters.DeviceIoControl.Type3InputBuffer** in the driver's **IO_STACK_LOCATION** structure, and the output buffer's address is specified by **Irp->UserBuffer**.

Buffer sizes are supplied by **Parameters.DeviceIoControl.InputBufferLength** and **Parameters.DeviceIoControl.OutputBufferLength** in the driver's **IO_STACK_LOCATION** structure.

For more information about the **CTL_CODE** macro and the transfer types listed above, see Defining I/O Control Codes.

Security Issues for I/O Control Codes

6/25/2019 • 2 minutes to read • Edit Online

Secure processing of IRPs that contain I/O control codes depends on defining IOCTL codes properly and on carefully examining parameters that the driver receives with the IRP.

When defining new IOCTL codes, use the following rules:

- Always specify a *FunctionCode* value that is equal to or greater than 0x800.
- Always specify a *RequiredAccess* value. The I/O manager does not send IOCTLs if the caller has insufficient access rights.
- Do not define IOCTL codes that allow callers to read or write nonspecific areas of kernel memory.

When processing IOCTL codes within a driver, use the following rules:

- Whenever a driver's dispatch routines test received IOCTL codes, they must always test the entire 32-bit value.
- Drivers can use **IoValidateDeviceIoControlAccess** to dynamically perform stricter access checking than that specified by the *RequiredAccess* value in the definition of the I/O control code.
- Never read or write more data than the buffer that is pointed to by **Irp->AssociatedIrp.SystemBuffer** can contain. Therefore, always check **Parameters.DeviceIoControl.InputBufferLength** or **Parameters.DeviceIoControl.OutputBufferLength** in the **IO_STACK_LOCATION** structure to determine buffer limits.
- Always zero driver-allocated buffers that will contain data intended for an application that originated an IOCTL request. That way, you will not accidentally copy sensitive data to the application.
- For METHOD_IN_DIRECT and METHOD_OUT_DIRECT transfers, follow the rules above. Additionally, check for a **NULL** return value from **MmGetSystemAddressForMdlSafe**, which indicates that mapping failed or that a zero-length buffer was supplied.
- For METHOD_NEITHER transfers, follow the rules that are provided in Using Neither Buffered Nor Direct I/O.

Using IRP Priority Hints

6/25/2019 • 2 minutes to read • Edit Online

An *IRP priority hint* is an **IO_PRIORITY_HINT** value that is associated with an IRP. IRP priority hints provide a simple hinting mechanism to indicate the relative importance of IRPs. A driver can use the priority hint for an IRP when choosing the order that the IRP is processed. IRP priority hints are available on Windows Vista and later operating systems.

For more information about IRP priority hints, see the I/O Prioritization in Windows Vista white paper.

Processing IRPs in a Lowest-Level Driver

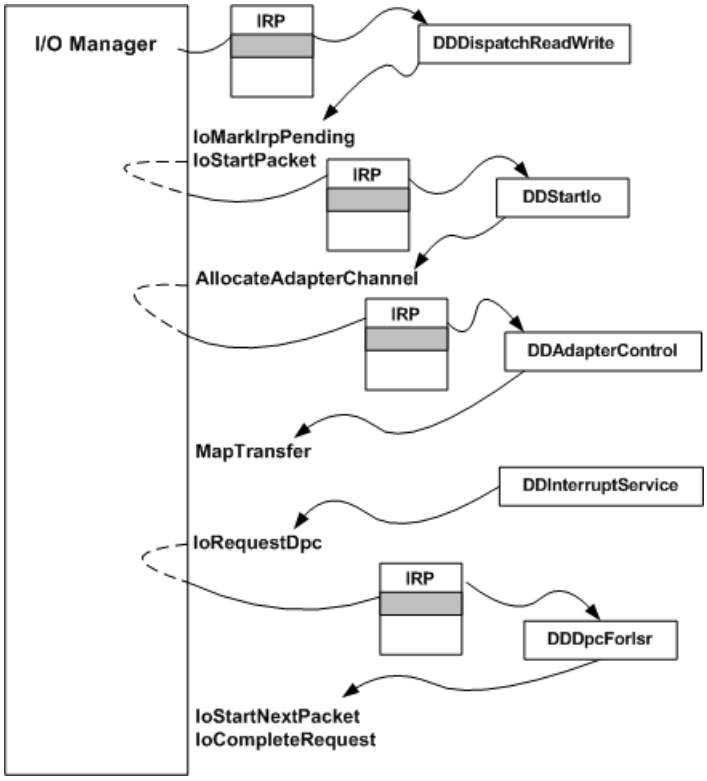
6/25/2019 • 5 minutes to read • Edit Online

Lowest-level physical drivers have certain standard routines that higher-level drivers do not need. The set of standard routines for lowest-level drivers also varies according to the following criteria:

- The nature of the device each driver controls
- Whether the driver sets up its device objects for direct or buffered I/O
- The design of the individual driver

To illustrate the roles of the standard driver routines, the following figure shows the path a sample IRP might take as it is processed by a lowest-level mass-storage device driver. The driver in the figure has the following characteristics:

- The device generates interrupts at the end of each I/O operation, so this driver has ISR and *DpcForIsr* routines.
- The driver has a *StartIo* routine, rather than setting up internal queues for IRPs and managing its own queuing.
- The driver uses system DMA, so it sets its device objects' **Flags** for direct I/O, and has an *AdapterControl* routine.



As this figure shows, the I/O manager creates an IRP and sends it to the driver's dispatch routine for the given major function code. Assuming the function code is either **IRP_MJ_READ** or **IRP_MJ_WRITE**, the dispatch routine is **DDDispatchReadWrite**.

Calling IoGetCurrentIrpStackLocation

Any driver routine that requires IRP parameters must call **IoGetCurrentIrpStackLocation** to obtain the driver's I/O stack location. Such routines include dispatch routines that handle more than one major I/O function code

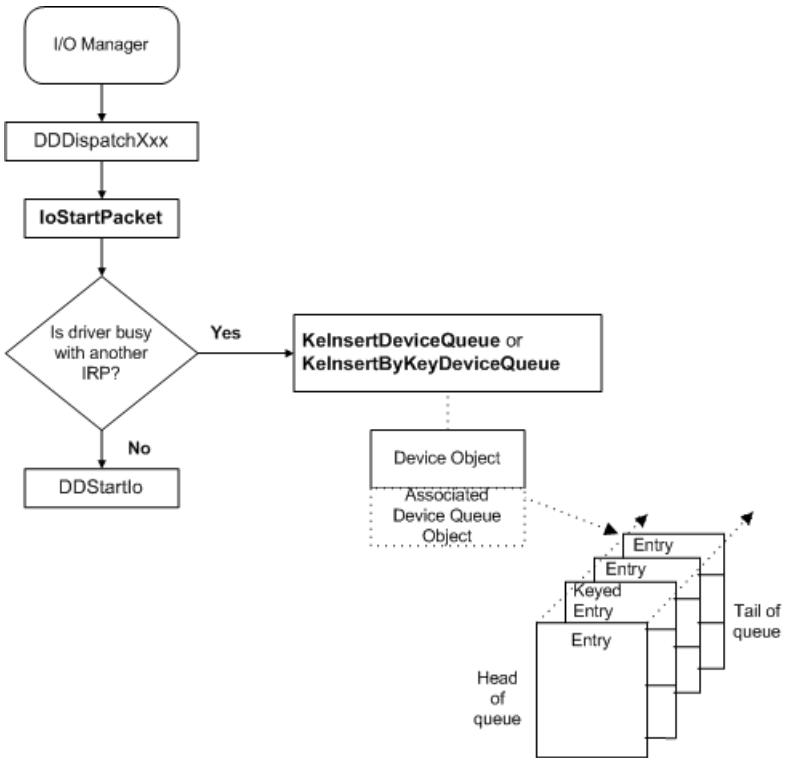
(**IRP_MJ_*XXX***), *handle a function that supports minor functions (IRP_MN_XXX**), or handle device I/O control requests (*IRP_MJ_DEVICE_CONTROL** and/or **IRP_MJ_INTERNAL_DEVICE_CONTROL**), along with every other driver routine that processes an IRP.

This driver's I/O stack location is the lowest one, with an indefinite number of higher-level drivers' I/O stack locations shown shaded. For simplicity, calls to **IoGetCurrentIrpStackLocation** from the *DispatchReadWrite*, *StartIo*, *AdapterControl*, and *DpcForIsr* routines are not shown in the previous figure.

Calling IoMarkIrpPending and IoStartPacket

The sample driver does not complete the IRP in its dispatch routine, but instead processes the IRP in its *StartIo* routine. Before it can do so, the dispatch routine calls **IoMarkIrpPending** to indicate that the IRP is not yet completed. Then it calls **IoStartPacket** to queue the IRP for further processing by the driver's *StartIo* routine. The dispatch routine also returns the NTSTATUS value STATUS_PENDING.

The following figure illustrates the call to **IoStartPacket**.



If the driver is busy processing another IRP on the device, **IoStartPacket** inserts the IRP into the device queue associated with the device object. The driver can optionally supply a *Key* value as a parameter to **IoStartPacket** to impose a driver-determined order on IRPs in the device queue.

If the driver is not busy and the device queue is empty, the I/O manager immediately calls its *StartIo* routine, passing the input IRP.

For mass-storage devices, the lowest-level driver does not need to supply a *Cancel* routine when it calls **IoStartPacket** for two reasons:

1. A file system layered over such a driver typically handles the cancellation of file I/O requests.
2. Mass-storage device drivers process IRPs quickly.

Usually, the highest-level driver in a chain of layered drivers handles the cancellation of IRPs.

Calling AllocateAdapterChannel and MapTransfer

Assuming the *StartIo* routine, shown in the figure illustrating an IRP path through lowest-level driver routines, determines that the transfer request can be done by a single DMA operation, the *StartIo* routine calls **AllocateAdapterChannel** with the entry point of the driver's *AdapterControl* routine and the IRP.

When the system DMA controller is available, the I/O manager calls the driver's *AdapterControl* routine to set up the transfer operation. The *AdapterControl* routine calls **MapTransfer** to set up the system DMA controller. Then the driver programs its device for the DMA operation and returns. (For more information about using DMA and adapter objects, see Input/Output Techniques.)

Calling IoRequestDpc from the Driver's ISR

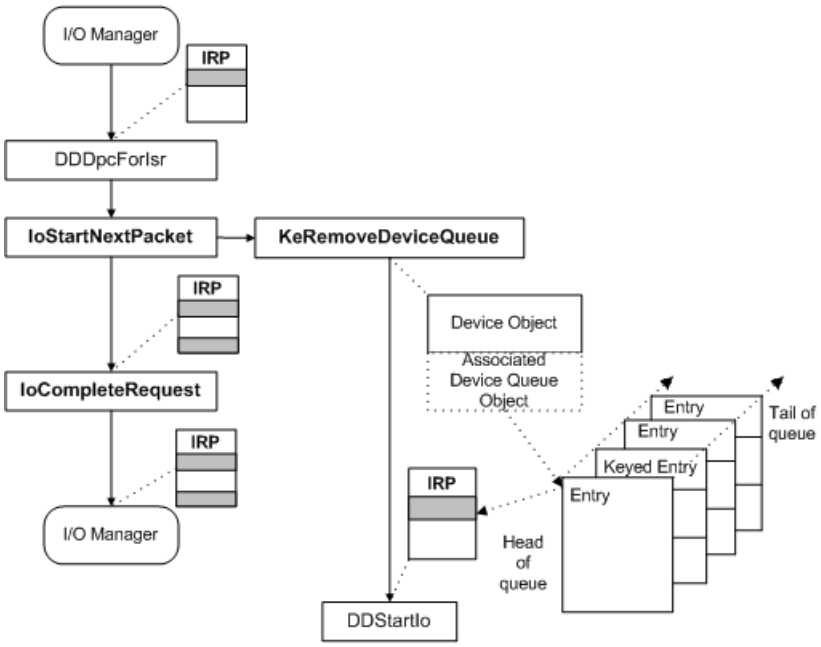
When the device interrupts to indicate its transfer operation is complete, the driver's ISR stops the device from generating interrupts and calls **IoRequestDpc**, as shown in the figure illustrating an IRP path through lowest-level driver routines.

This call queues the driver's *DpcForIsr* routine to complete as much of the transfer operation as possible at a lower hardware priority (IRQL).

Calling IoStartNextPacket and IoCompleteRequest

When the *DpcForIsr* routine has finished processing the transfer, it calls **IoStartNextPacket** promptly so that the driver's *StartIo* routine will be called with the next IRP in the device queue, if any are queued. The *DpcForIsr* routine also sets the just-completed IRP's I/O status block and then calls **IoCompleteRequest** for the IRP.

The following figure illustrates this driver's calls to **IoStartNextPacket** and **IoCompleteRequest**.



Drivers should call **IoStartNextPacket** or **IoStartNextPacketByKey** to begin the next requested I/O operation as soon as possible, preferably before they call **IoCompleteRequest**.

If any IRPs are queued for the device, **IoStartNextPacket** calls **KeRemoveDeviceQueue** to remove the next IRP from the queue. The I/O manager then calls the driver's *StartIo* routine, passing the dequeued IRP. If no IRPs are currently in the device queue, **IoStartNextPacket** merely returns to the caller.

Setting the I/O Status Block in an IRP

Every lowest-level driver must set the IRP's I/O status block before calling **IoCompleteRequest**. (In the previous figure, the second shaded area denotes the status block.) The I/O status block supplies information to higher-level drivers and, ultimately, to the original requester of the I/O operation. Any higher-level driver layered above the driver in the previous figure might have set up an *IoCompletion* routine that reads the I/O status block set by this driver. Higher-level drivers usually do not modify the I/O status block in an IRP that has been completed by a device driver, unless the higher-level driver is retrying the IRP, in which case it reinitializes the I/O status block.

Every higher-level driver that completes an IRP without sending it on to the next lower driver also must set the I/O status block in that IRP before calling **IoCompleteRequest**. For good overall I/O throughput, a higher-level driver should check the parameters in its own I/O stack location of each IRP and, if the parameters are invalid, should set

the I/O status block and complete the request itself. Whenever possible, a driver should avoid passing an invalid request on to lower drivers in the chain.

Assuming the transfer operation in the previous figure is successful, the *DpcForIsr* routine, shown in the figure illustrating an IRP path through lowest-level driver routines, sets STATUS_SUCCESS in **Status** and the number of bytes transferred in **Information** for the IRP's I/O status block.

Many of the standard driver routines also return NTSTATUS-type values. For more information about NTSTATUS constants like STATUS_SUCCESS, see Logging Errors.

Processing IRPs in an Intermediate-Level Driver

6/25/2019 • 4 minutes to read • Edit Online

Higher-level drivers have a different set of standard routines than lowest-level device drivers, with an overlapping subset of standard routines common to both types of drivers.

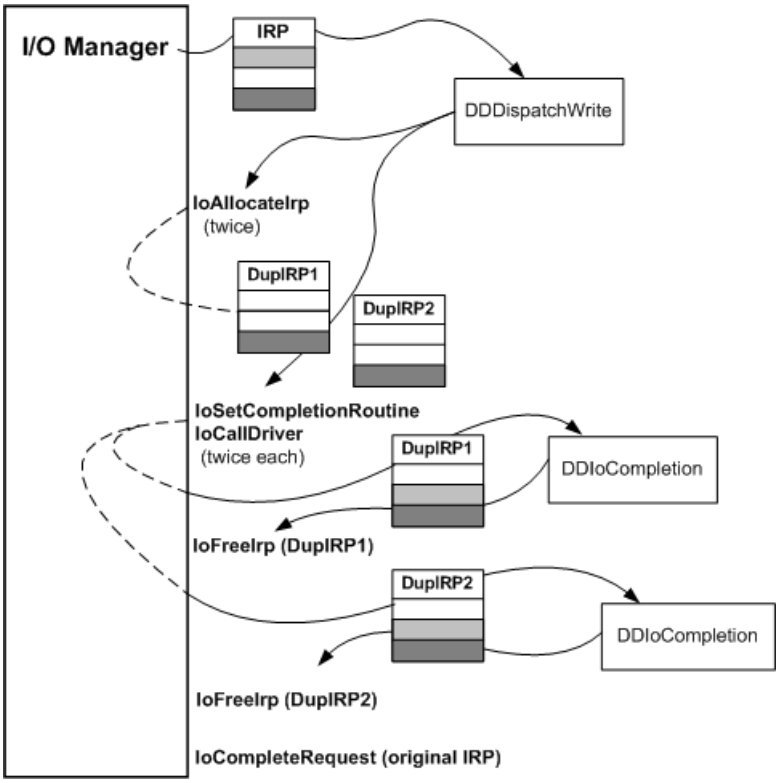
The set of routines for intermediate and highest-level drivers also varies according to the following criteria:

- The nature of the underlying physical device
- Whether an underlying device driver sets up device objects for direct or buffered I/O
- The design of the individual higher-level driver

The following figure illustrates the path an IRP might take through the standard routines of an intermediate *mirror driver* layered somewhere over the lowest-level device driver described in the previous section.

The driver shown in the following figure has the following characteristics:

- The driver is layered over more than one physical device and possibly over more than one device driver.
- The driver sometimes allocates additional IRPs for lower-level drivers, depending on the requested operation in the input IRP.
- The driver has at least one file system driver layered above it, and that file system driver might be layered over other intermediate drivers at a higher level than this one.



As the figure shows, the I/O manager creates an IRP and sends it to the driver's dispatch routine for the given major function code. Assuming the function code is **IRP_MJ_WRITE**, the dispatch routine is **DDDispatchWrite**. The intermediate driver's I/O stack location is shown in the middle, with an indefinite number of I/O stack locations for higher- and lower-level drivers shown shaded.

Allocating IRPs

The mirror driver's purpose is to send write requests to several physical devices, and to send read requests alternately to the drivers of these devices. For write requests, the driver creates duplicate IRPs for each device on which the data is to be written, assuming the parameters in the input IRP are valid.

The previous figure shows a call to **IoAllocateIrp** but higher-level drivers can call other support routines to allocate IRPs for lower-level drivers. See Creating IRPs for Lower-Level Drivers.

When the dispatch routine calls **IoAllocateIrp**, it specifies the number of I/O stack locations needed for the IRP. The driver must specify a stack location for each lower driver in the chain, getting the appropriate value from the device objects of each driver just below the mirror driver. Optionally, the driver can add one to this value when it calls **IoAllocateIrp** to get a stack location of its own for each IRP it allocates, as the driver in the previous figure does.

This intermediate driver's dispatch routine calls **IoGetCurrentIrpStackLocation** (not shown) with the original IRP, to check parameters.

It calls **IoSetNextIrpStackLocation** because it allocated its own stack location in each newly created IRP and **IoGetCurrentIrpStackLocation** to create a context for itself that it uses later in the *IoCompletion* routine.

Next, it calls **IoGetNextIrpStackLocation** with each newly created IRP so that it can set up the next lower-level drivers' I/O stack locations in the IRPs it allocated. The mirror driver's dispatch routine copies the IRP function codes and parameters (pointer to the transfer buffer, length in bytes to be transferred for **IRP_MJ_WRITE**) into the I/O stack locations for the next-lower drivers. These drivers, in turn, will set up the I/O stack locations for the drivers just below them, if any.

Calling IoSetCompletionRoutine and IoCallDriver

The dispatch routine in the previous figure calls **IoSetCompletionRoutine** for each IRP it allocated. Because the driver in the previous figure must dispose of the IRPs it allocated, this driver sets its *IoCompletion* routine to be called when lower drivers complete its IRPs, whether the I/O operation completed successfully, failed, or was canceled.

Because the driver in the previous figure mirrors in parallel, it passes both IRPs that it allocated on to the next-lower-level drivers by calling **IoCallDriver** twice, once for each target device object representing a mirrored partition.

Processing IRPs in the Driver's IoCompletion Routine

When either set of lower-level drivers completes the requested operation, the I/O manager calls the intermediate mirror driver's *IoCompletion* routine. The mirror driver maintains a count in its own I/O stack location for the original IRP, to track when the lower drivers have completed all the duplicate IRPs.

Assuming that the I/O status block indicates that one set of lower drivers has completed the duplicate IRP shown in the previous figure, the mirror driver's *IoCompletion* routine decrements its count but cannot complete the original IRP until it decrements the count to zero. If the decremented count is not yet zero, the *IoCompletion* routine calls **IoFreeIrp** with the first-returned IRP (DupIRP1 in the previous figure) that the driver allocated and returns STATUS_MORE_PROCESSING_REQUIRED.

When the mirror driver's *IoCompletion* routine is called again with the DupIRP2 shown in the previous figure, the *IoCompletion* routine decrements the count in the original IRP and determines that both sets of lower-level drivers have carried out the requested operations.

Assuming the I/O status block in DupIRP2 also is set with STATUS_SUCCESS, the *IoCompletion* routine copies the I/O status block from DupIRP2 into the original IRP and frees DupIRP2. It calls **IoCompleteRequest** with the original IRP and returns STATUS_MORE_PROCESSING_REQUIRED. Returning this status prevents the I/O manager from attempting any further completion processing on DupIRP2; because the IRP is not associated with a thread, its completion processing should end with the driver that created it.

If either set of lower-level drivers does not complete the mirror driver's IRPs successfully, the mirror driver's

IoCompletion routine should log an error and attempt appropriate mirrored-data recovery. For more information, see Logging Errors.

Different ways of handling IRPs - Cheat sheet

6/25/2019 • 23 minutes to read • Edit Online

A Windows Driver Model (WDM) driver typically sends input/output request packets (IRPs) to other drivers. A driver either creates its own IRP and sends it to a lower driver, or the driver forwards the IRPs that it receives from another driver that is attached above.

This article discusses different ways that a driver can send IRPs to a lower driver and includes annotated sample code.

- Scenarios 1-5 are about how to forward an IRP to a lower driver from a dispatch routine.
- Scenarios 6-12 discuss different ways of creating an IRP and sending it to another driver.

Before you examine the various scenarios, note that an IRP completion routine can return either STATUS_MORE_PROCESSING_REQUIRED or STATUS_SUCCESS.

The I/O manager uses the following rules when it examines the status:

- If the status is STATUS_MORE_PROCESSING_REQUIRED, stop completing the IRP, leave the stack location unchanged and return.
- If the status is anything other than STATUS_MORE_PROCESSING_REQUIRED, continue completing the IRP upward.

Because the I/O Manager does not have to know which non-STATUS_MORE_PROCESSING_REQUIRED value is used, use STATUS_SUCCESS (because the value 0 loads efficiently on most processor architectures).

As you read the following code, note that STATUS_CONTINUE_COMPLETION is aliased to STATUS_SUCCESS in the WDK.

```
// This value should be returned from completion routines to continue
// completing the IRP upwards. Otherwise, STATUS_MORE_PROCESSING_REQUIRED
// should be returned.
//
#define STATUS_CONTINUE_COMPLETION      STATUS_SUCCESS
//
// Completion routines can also use this enumeration instead of status codes.
//
typedef enum _IO_COMPLETION_ROUTINE_RESULT {

    ContinueCompletion = STATUS_CONTINUE_COMPLETION,
    StopCompletion = STATUS_MORE_PROCESSING_REQUIRED

} IO_COMPLETION_ROUTINE_RESULT, *PIO_COMPLETION_ROUTINE_RESULT;
```

Forwarding an IRP to another driver

Scenario 1: Forward and forget

Use the following code if a driver just wants to forward the IRP down and take no additional action. The driver does not have to set a completion routine in this case. If the driver is a top level driver, the IRP can be completed synchronously or asynchronously, depending on the status that is returned by the lower driver.

```
NTSTATUS
DispatchRoutine_1(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    //
    // You are not setting a completion routine, so just skip the stack
    // location because it provides better performance.
    //
    IoSkipCurrentIrpStackLocation (Irp);
    return IoCallDriver(TopOfDeviceStack, Irp);
}
```

Scenario 2: Forward and wait

Use the following code if a driver wants to forward the IRP to a lower driver and wait for it to return so that it can process the IRP. This is frequently done when handling PNP IRPs. For example, when you receive a IRP_MN_START_DEVICE IRP, you must forward the IRP down to the bus driver and wait for it to complete before you can start your device. You can call **IoForwardIrpSynchronously** to do this operation easily.

```
NTSTATUS
DispatchRoutine_2(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    KEVENT   event;
    NTSTATUS status;

    KeInitializeEvent(&event, NotificationEvent, FALSE);

    //
    // You are setting completion routine, so you must copy
    // current stack location to the next. You cannot skip a location
    // here.
    //
    IoCopyCurrentIrpStackLocationToNext(Irp);

    IoSetCompletionRoutine(Irp,
                           CompletionRoutine_2,
                           &event,
                           TRUE,
                           TRUE,
                           TRUE
                           );

    status = IoCallDriver(TopOfDeviceStack, Irp);

    if (status == STATUS_PENDING) {

       KeWaitForSingleObject(&event,
                             Executive, // WaitReason
                             KernelMode, // must be Kernelmode to prevent the stack getting paged out
                             FALSE,
                             NULL // indefinite wait
                             );
       status = Irp->IoStatus.Status;
    }

    // <---- Do your own work here.


    //
    // Because you stopped the completion of the IRP in the CompletionRoutine
```

```
    // by returning STATUS_MORE_PROCESSING_REQUIRED, you must call
    // IoCompleteRequest here.
    //
    IoCompleteRequest (Irp, IO_NO_INCREMENT);
    return status;

}
NTSTATUS
CompletionRoutine_2(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
  if (Irp->PendingReturned == TRUE) {
    //
    // You will set the event only if the lower driver has returned
    // STATUS_PENDING earlier. This optimization removes the need to
    // call KeSetEvent unnecessarily and improves performance because the
    // system does not have to acquire an internal lock.
    //
    KeSetEvent ((PKEVENT) Context, IO_NO_INCREMENT, FALSE);
  }
  // This is the only status you can return.
  return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Scenario 3: Forward with a completion routine

In this case, the driver sets a completion routine, forwards the IRP down, and then returns the status of lower driver as is. The purpose of setting the completion routine is to modify the content of the IRP on its way back.

```
NTSTATUS
DispathRoutine_3(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    NTSTATUS status;

    //
    // Because you are setting completion routine, you must copy the
    // current stack location to the next. You cannot skip a location
    // here.
    //
    IoCopyCurrentIrpStackLocationToNext(Irp);

    IoSetCompletionRoutine(Irp,
                           CompletionRoutine_31,// or CompletionRoutine_32
                           NULL,
                           TRUE,
                           TRUE,
                           TRUE
                           );

    return IoCallDriver(TopOfDeviceStack, Irp);
}
```

If you return the status of the lower driver from your dispatch routine:

- You must not change the status of the IRP in the completion routine. This is to make sure that the status values set in the IRP's IoStatus block (Irp->IoStatus.Status) are the same as the return status of the lower drivers.
- You must propagate the pending status of the IRP as indicated by Irp->PendingReturned.
- You must not change the synchronicity of the IRP.

As a result, there are only 2 valid versions of the completion routine in this scenario (31 and 32):

```
NTSTATUS
CompletionRoutine_31 (
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{

    //
    // Because the dispatch routine is returning the status of lower driver
    // as is, you must do the following:
    //
    if (Irp->PendingReturned) {

        IoMarkIrpPending( Irp );
    }

    return STATUS_CONTINUE_COMPLETION ; // Make sure of same synchronicity
}

NTSTATUS
CompletionRoutine_32 (
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    //
    // Because the dispatch routine is returning the status of lower driver
    // as is, you must do the following:
    //
    if (Irp->PendingReturned) {

        IoMarkIrpPending( Irp );
    }

    //
    // To make sure of the same synchronicity, complete the IRP here.
    // You cannot complete the IRP later in another thread because the
    // the dispatch routine is returning the status returned by the lower
    // driver as is.
    //
    IoCompleteRequest( Irp,  IO_NO_INCREMENT);

    //
    // Although this is an unusual completion routine that you rarely see,
    // it is discussed here to address all possible ways to handle IRPs.
    //
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Scenario 4: Queue for later, or forward and reuse

Use the following code snippet in a situation where the driver wants to either queue an IRP and process it later or forward the IRP to the lower driver and reuse it for a specific number of times before completing the IRP. The dispatch routine marks the IRP pending and returns STATUS_PENDING because the IRP is going to be completed later in a different thread. Here, the completion routine can change the status of the IRP if necessary (in contrast to the previous scenario).

```
NTSTATUS
DispathRoutine_4(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    NTSTATUS status;

    //
    // You mark the IRP pending if you are intending to queue the IRP
    // and process it later. If you are intending to forward the IRP
    // directly, use one of the methods discussed earlier in this article.
    //
    IoMarkIrpPending( Irp );

    //
    // For demonstration purposes: this IRP is forwarded to the lower driver.
    //
    IoCopyCurrentIrpStackLocationToNext(Irp);

    IoSetCompletionRoutine(Irp,
                           CompletionRoutine_41, // or CompletionRoutine_42
                           NULL,
                           TRUE,
                           TRUE,
                           TRUE
                           );
    IoCallDriver(TopOfDeviceStack, Irp);

    //
    // Because you marked the IRP pending, you must return pending,
    // regardless of the status of returned by IoCallDriver.
    //
    return STATUS_PENDING ;

}
```

The completion routine can either return STATUS_CONTINUE_COMPLETION or STATUS_MORE_PROCESSING_REQUIRED. You return STATUS_MORE_PROCESSING_REQUIRED only if you intend to reuse the IRP from another thread and complete it later.

```
NTSTATUS
CompletionRoutine_41(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    //
    // By returning STATUS_CONTINUE_COMPLETION , you are relinquishing the
    // ownership of the IRP. You cannot touch the IRP after this.
    //
    return STATUS_CONTINUE_COMPLETION ;
}


NTSTATUS
CompletionRoutine_42 (
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    //
    // Because you are stopping the completion of the IRP by returning the
    // following status, you must complete the IRP later.
    //
    return STATUS_MORE_PROCESSING_REQUIRED ;
}
```

Scenario 5: Complete the IRP in the dispatch routine

This scenario shows how to complete an IRP in the dispatch routine.

Important When you complete an IRP in the dispatch routine, the return status of the dispatch routine should match the status of the value that is set in the IoStatus block of the IRP (Irp->IoStatus.Status).

```
NTSTATUS
DispatchRoutine_5(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    //
    // <-- Process the IRP here.
    //
    Irp->IoStatus.Status = STATUS_XXX;
    Irp->IoStatus.Information = YYY;
    IoCompletRequest(Irp, IO_NO_INCREMENT);
    return STATUS_XXX;
}
```

Creating IRPs and sending them to another driver

Introduction

Before you examine the scenarios, you must understand the differences between a driver-created synchronous input/output request packet (IRP) and an asynchronous request.

SYNCHRONOUS (THREADED) IRP	ASYNCHRONOUS (NON-THREADED) IRP

SYNCHRONOUS (THREADED) IRP	ASYNCHRONOUS (NON-THREADED) IRP
Created by using IoBuildSynchronousFsdRequest or IoBuildDeviceIoControlRequest .	Created by using IoBuildAsynchronousFsdRequest or IoAllocateIrp . This is meant for driver to driver communication.
Thread must wait for the IRP to complete.	Thread does not have to wait for the IRP to complete.
Associated with the thread that created it, hence, the name threaded IRPs. Therefore, if the thread exits, the I/O manager cancels the IRP.	Not associated with the thread that created it.
Cannot be created in an arbitrary thread context.	Can be created in an arbitrary thread context because the thread does not wait for the IRP to complete.
The I/O manager does the post completions to free the buffer that is associated with the IRP.	The I/O manager cannot do the cleanup. The driver must provide a completion routine and free the buffers that are associated with the IRP.
Must be sent at IRQL level equal to PASSIVE_LEVEL.	Can be sent at IRQL less than or equal to DISPATCH_LEVEL if the Dispatch routine of the target driver can handle the request at DISPATCH_LEVEL.

Scenario 6: Send a synchronous device-control request (IRP_MJ_INTERNAL_DEVICE_CONTROL/IRP_MJ_DEVICE_CONTROL) by using IoBuildDeviceIoControlRequest

The following code shows how to call **IoBuildDeviceIoControlRequest** request to make a synchronous IOCTL request. For more info, see IRP_MJ_INTERNAL_DEVICE_CONTROL and IRP_MJ_DEVICE_CONTROL.

```
NTSTATUS
MakeSynchronousIoctl(
    IN PDEVICE_OBJECT    TopOfDeviceStack,
    IN ULONG         IoctlControlCode,
    PVOID             InputBuffer,
    ULONG             InputBufferLength,
    PVOID             OutputBuffer,
    ULONG             OutputBufferLength
    )
/*++

Arguments:

    TopOfDeviceStack-

    IoctlControlCode              - Value of the IOCTL request

    InputBuffer        - Buffer to be sent to the TopOfDeviceStack

    InputBufferLength  - Size of buffer to be sent to the TopOfDeviceStack

    OutputBuffer       - Buffer for received data from the TopOfDeviceStack

    OutputBufferLength - Size of receive buffer from the TopOfDeviceStack

Return Value:

    NT status code

--*/
{
    KEVENT              event;
    PIRP                irp;
```

```
    IO_STATUS_BLOCK     ioStatus;
    NTSTATUS status;

    //
    // Creating Device control IRP and send it to the another
    // driver without setting a completion routine.
    //

    KeInitializeEvent(&event, NotificationEvent, FALSE);

    irp = IoBuildDeviceIoControlRequest (
                            IoctlControlCode,
                            TopOfDeviceStack,
                            InputBuffer,
                            InputBufferLength,
                            OutputBuffer,
                            OutputBufferLength,
                            FALSE, // External
                            &event,
                            &ioStatus);

    if (NULL == irp) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }


    status = IoCallDriver(TopOfDeviceStack, irp);

    if (status == STATUS_PENDING) {
        //
        // You must wait here for the IRP to be completed because:
        // 1) The IoBuildDeviceIoControlRequest associates the IRP with the
        //     thread and if the thread exits for any reason, it would cause the IRP
        //     to be canceled.
        // 2) The Event and IoStatus block memory is from the stack and we
        //     cannot go out of scope.
        // This event will be signaled by the I/O manager when the
        // IRP is completed.
        //
        status = KeWaitForSingleObject(
                    &event,
                    Executive, // wait reason
                    KernelMode, // To prevent stack from being paged out.
                    FALSE,     // You are not alertable
                    NULL);     // No time out !!!!

        status = ioStatus.Status;
    }

    return status;
}
```

Scenario 7: Send a synchronous device-control (IOCTL) request and cancel it if not completed in a certain time period

This scenario is similar to the previous scenario except that instead of waiting indefinitely for the request to complete, it waits for some user-specified time and safely cancels the IOCTL request if the wait times out.

```
typedef enum {

   IRPLOCK_CANCELABLE,
   IRPLOCK_CANCEL_STARTED,
   IRPLOCK_CANCEL_COMPLETE,
   IRPLOCK_COMPLETED

} IRPLOCK;
//
```

```
//
// An IRPLOCK allows for safe cancellation. The idea is to protect the IRP
// while the canceller is calling IoCancelIrp. This is done by wrapping the
// call in InterlockedExchange(s). The roles are as follows:
//
// Initiator/completion: Cancelable --> IoCallDriver() --> Completed
// Canceller: CancelStarted --> IoCancelIrp() --> CancelCompleted
//
// No cancellation:
//   Cancelable-->Completed
//
// Cancellation, IoCancelIrp returns before completion:
//   Cancelable --> CancelStarted --> CancelCompleted --> Completed
//
// Canceled after completion:
//   Cancelable --> Completed -> CancelStarted
//
// Cancellation, IRP completed during call to IoCancelIrp():
//   Cancelable --> CancelStarted -> Completed --> CancelCompleted
//
//  The transition from CancelStarted to Completed tells the completer to block
//  postprocessing (IRP ownership is transferred to the canceller). Similarly,
//  the canceller learns it owns IRP postprocessing (free, completion, etc)
//  during a Completed->CancelCompleted transition.
//


NTSTATUS
MakeSynchronousIoctlWithTimeOut(
    IN PDEVICE_OBJECT    TopOfDeviceStack,
    IN ULONG         IoctlControlCode,
    PVOID             InputBuffer,
    ULONG             InputBufferLength,
    PVOID             OutputBuffer,
    ULONG             OutputBufferLength,
    IN  ULONG               Milliseconds
    )
/*++

Arguments:

    TopOfDeviceStack   -

    IoctlControlCode   - Value of the IOCTL request.

    InputBuffer        - Buffer to be sent to the TopOfDeviceStack.

    InputBufferLength  - Size of buffer to be sent to the TopOfDeviceStack.

    OutputBuffer       - Buffer for received data from the TopOfDeviceStack.

    OutputBufferLength - Size of receive buffer from the TopOfDeviceStack.

    Milliseconds       - Timeout value in Milliseconds

Return Value:

    NT status code

--*/
{
    NTSTATUS status;
    PIRP irp;
    KEVENT event;
    IO_STATUS_BLOCK ioStatus;
    LARGE_INTEGER dueTime;
    IRPLOCK lock;

    KeInitializeEvent(&event, NotificationEvent, FALSE);
```

```
irp = IoBuildDeviceIoControlRequest (
                IoctlControlCode,
                TopOfDeviceStack,
                InputBuffer,
                InputBufferLength,
                OutputBuffer,
                OutputBufferLength,
                FALSE, // External ioctl
                &event,
                &ioStatus);



if (irp == NULL) {
    return STATUS_INSUFFICIENT_RESOURCES;
}

lock = IRPLOCK_CANCELABLE;

IoSetCompletionRoutine(
                irp,
                MakeSynchronousIoctlWithTimeOutCompletion,
                &lock,
                TRUE,
                TRUE,
                TRUE
                );

status = IoCallDriver(TopOfDeviceStack, irp);

if (status == STATUS_PENDING) {

    dueTime.QuadPart = -10000 * Milliseconds;

    status = KeWaitForSingleObject(
                        &event,
                        Executive,
                        KernelMode,
                        FALSE,
                        &dueTime
                        );

    if (status == STATUS_TIMEOUT) {

        if (InterlockedExchange((PVOID)&lock, IRPLOCK_CANCEL_STARTED) == IRPLOCK_CANCELABLE) {

            //
            // You got it to the IRP before it was completed. You can cancel
            // the IRP without fear of losing it, because the completion routine
            // does not let go of the IRP until you allow it.
            //
            IoCancelIrp(irp);

            //
            // Release the completion routine. If it already got there,
            // then you need to complete it yourself. Otherwise, you got
            // through IoCancelIrp before the IRP completed entirely.
            //
            if (InterlockedExchange(&lock, IRPLOCK_CANCEL_COMPLETE) == IRPLOCK_COMPLETED) {
                IoCompleteRequest(irp, IO_NO_INCREMENT);
            }
        }

        KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);

        ioStatus.Status = status; // Return STATUS_TIMEOUT

    } else {
```

```
            status = ioStatus.Status;
        }
    }

    return status;
}

NTSTATUS
MakeSynchronousIoctlWithTimeOutCompletion(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    PLONG lock;

    lock = (PLONG) Context;

    if (InterlockedExchange(lock, IRPLOCK_COMPLETED) == IRPLOCK_CANCEL_STARTED) {
        //
        // Main line code has got the control of the IRP. It will
        // now take the responsibility of completing the IRP.
        // Therefore...
        return STATUS_MORE_PROCESSING_REQUIRED;
    }

    return STATUS_CONTINUE_COMPLETION ;
}
```

Scenario 8: Send a synchronous non-IOCTL request by using IoBuildSynchronousFsdRequest - completion routine returns STATUS_CONTINUE_COMPLETION

The following code shows how to make a synchronous non-IOCTL request by calling **IoBuildSynchronousFsdRequest**. The technique shown here is similar to scenario 6.

```
NTSTATUS
MakeSynchronousNonIoctlRequest (
    PDEVICE_OBJECT   TopOfDeviceStack,
    PVOID               WriteBuffer,
    ULONG               NumBytes
    )
/*++
Arguments:

    TopOfDeviceStack -

    WriteBuffer       - Buffer to be sent to the TopOfDeviceStack.

    NumBytes  - Size of buffer to be sent to the TopOfDeviceStack.

Return Value:

    NT status code


--*/
{
    NTSTATUS        status;
    PIRP            irp;
    LARGE_INTEGER   startingOffset;
    KEVENT          event;
    IO_STATUS_BLOCK     ioStatus;
    PVOID context;

    startingOffset.QuadPart = (LONGLONG) 0;
    //
    // Allocate memory for any context information to be passed
```

```
    // Allocate memory for any context information to be passed
    // to the completion routine.
    //
    context = ExAllocatePoolWithTag(NonPagedPool, sizeof(ULONG), 'ITag');
    if(!context) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    KeInitializeEvent(&event,  NotificationEvent,   FALSE);

    irp = IoBuildSynchronousFsdRequest(
                IRP_MJ_WRITE,
                TopOfDeviceStack,
                WriteBuffer,
                NumBytes,


                &startingOffset, // Optional
                &event,
                &ioStatus
                );

    if (NULL == irp) {
        ExFreePool(context);
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    IoSetCompletionRoutine(irp,
                   MakeSynchronousNonIoctlRequestCompletion,
                   context,
                   TRUE,
                   TRUE,
                   TRUE);

    status = IoCallDriver(TopOfDeviceStack, irp);

    if (status == STATUS_PENDING) {

       status = KeWaitForSingleObject(
                           &event,
                           Executive,
                           KernelMode,
                           FALSE, // Not alertable
                           NULL);
        status = ioStatus.Status;
    }

    return status;
}
NTSTATUS
MakeSynchronousNonIoctlRequestCompletion(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    if (Context) {
        ExFreePool(Context);
    }
    return STATUS_CONTINUE_COMPLETION ;

}
```

Scenario 9: Send a synchronous non-IOCTL request by using IoBuildSynchronousFsdRequest - completion routine returns STATUS_MORE_PROCESSING_REQUIRED

The only difference between this scenario and scenario 8 is that the completion routine returns STATUS_MORE_PROCESSING_REQUIRED.

```
NTSTATUS MakeSynchronousNonIoctlRequest2(
    PDEVICE_OBJECT TopOfDeviceStack,
    PVOID WriteBuffer,
    ULONG NumBytes
    )
/*++ Arguments:
    TopOfDeviceStack

    WriteBuffer     - Buffer to be sent to the TopOfDeviceStack.

    NumBytes        - Size of buffer to be sent to the TopOfDeviceStack.

Return Value:
    NT status code
--*/
{
    NTSTATUS        status;
    PIRP            irp;
    LARGE_INTEGER   startingOffset;
    KEVENT          event;
    IO_STATUS_BLOCK ioStatus;
    BOOLEAN         isSynchronous = TRUE;

    startingOffset.QuadPart = (LONGLONG) 0;
    KeInitializeEvent(&event, NotificationEvent, FALSE);
    irp = IoBuildSynchronousFsdRequest(
                IRP_MJ_WRITE,
                TopOfDeviceStack,
                WriteBuffer,
                NumBytes,
                &startingOffset, // Optional
                &event,
                &ioStatus
                );

    if (NULL == irp) {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    IoSetCompletionRoutine(irp,
                MakeSynchronousNonIoctlRequestCompletion2,
                (PVOID)&event,
                TRUE,
                TRUE,
                TRUE);

    status = IoCallDriver(TopOfDeviceStack, irp);

    if (status == STATUS_PENDING) {

        KeWaitForSingleObject(&event,
                              Executive,
                              KernelMode,
                              FALSE, // Not alertable
                              NULL);
        status = irp->IoStatus.Status;
        isSynchronous = FALSE;
    }

    //
    // Because you have stopped the completion of the IRP, you must
    // complete here and wait for it to be completed by waiting
    // on the same event again, which will be signaled by the I/O
    // manager.
    // NOTE: you cannot queue the IRP for
    // reuse by calling IoReuseIrp because it does not break the
    // association of this IRP with the current thread.
    //
```

```
    KeClearEvent(&event);
    IoCompleteRequest(irp, IO_NO_INCREMENT);

    //
    // We must wait here to prevent the event from going out of scope.
    // I/O manager will signal the event and copy the status to our
    // IoStatus block for synchronous IRPs only if the return status is not
    // an error. For asynchronous IRPs, the above mentioned copy operation
    // takes place regardless of the status value.
    //

    if (!(NT_ERROR(status) && isSynchronous)) {
        KeWaitForSingleObject(&event,
                              Executive,
                              KernelMode,
                              FALSE, // Not alertable
                              NULL);
    }
    return status;
}

NTSTATUS MakeSynchronousNonIoctlRequestCompletion2(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context )
{
    if (Irp->PendingReturned) {
        KeSetEvent ((PKEVENT) Context, IO_NO_INCREMENT, FALSE);
    }
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Scenario 10: Send an asynchronous request by using IoBuildAsynchronousFsdRequest

This scenario shows how to make an asynchronous request by calling **IoBuildAsynchronousFsdRequest**.

In an asynchronous request, the thread that made the request does not have to wait for the IRP to complete. The IRP can be created in an arbitrary thread context because the IRP is not associated with the thread. You must provide a completion routine and release the buffers and IRP in the completion routine if you do not intend to reuse the IRP. This is because the I/O manager cannot do post-completion cleanup of driver-created asynchronous IRPs (created with **IoBuildAsynchronousFsdRequest** and **IoAllocateIrp**).

```
NTSTATUS
MakeAsynchronousRequest (
    PDEVICE_OBJECT   TopOfDeviceStack,
    PVOID               WriteBuffer,
    ULONG               NumBytes
    )
/*++
Arguments:

    TopOfDeviceStack -

    WriteBuffer       - Buffer to be sent to the TopOfDeviceStack.

    NumBytes  - Size of buffer to be sent to the TopOfDeviceStack.

--*/
{
    NTSTATUS        status;
    PIRP            irp;
    LARGE_INTEGER   startingOffset;
    PIO_STACK_LOCATION  nextStack;
    PVOID context;
```

```
    startingOffset.QuadPart = (LONGLONG) 0;

    irp = IoBuildAsynchronousFsdRequest(
                IRP_MJ_WRITE,
                TopOfDeviceStack,
                WriteBuffer,
                NumBytes,
                &startingOffset, // Optional
                NULL
                );

    if (NULL == irp) {

        return STATUS_INSUFFICIENT_RESOURCES;
    }

    //
    // Allocate memory for context structure to be passed to the completion routine.
    //
    context = ExAllocatePoolWithTag(NonPagedPool, sizeof(ULONG_PTR), 'ITag');
    if (NULL == context) {
        IoFreeIrp(irp);
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    IoSetCompletionRoutine(irp,
                   MakeAsynchronousRequestCompletion,
                   context,
                   TRUE,
                   TRUE,
                   TRUE);
    //
    // If you want to change any value in the IRP stack, you must
    // first obtain the stack location by calling IoGetNextIrpStackLocation.
    // This is the location that is initialized by the IoBuildxxx requests and
    // is the one that the target device driver is going to view.
    //
    nextStack = IoGetNextIrpStackLocation(irp);
    //
    // Change the MajorFunction code to something appropriate.
    //
    nextStack->MajorFunction = IRP_MJ_SCSI;

    (void) IoCallDriver(TopOfDeviceStack, irp);

    return STATUS_SUCCESS;
}
NTSTATUS
MakeAsynchronousRequestCompletion(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    PMDL mdl, nextMdl;

    //
    // If the target device object is set up to do buffered i/o
    // (TopOfDeviceStack->Flags and DO_BUFFERED_IO), then
    // IoBuildAsynchronousFsdRequest request allocates a system buffer
    // for read and write operation. If you stop the completion of the IRP
    // here, you must free that buffer.
    //

    if(Irp->AssociatedIrp.SystemBuffer && (Irp->Flags & IRP_DEALLOCATE_BUFFER) ) {
            ExFreePool(Irp->AssociatedIrp.SystemBuffer);
    }

    //
```

```
    //
    // If the target device object is set up do direct i/o (DO_DIRECT_IO), then
    // IoBuildAsynchronousFsdRequest creates an MDL to describe the buffer
    // and locks the pages. If you stop the completion of the IRP, you must unlock
    // the pages and free the MDL.
    //

    else if (Irp->MdlAddress != NULL) {
        for (mdl = Irp->MdlAddress; mdl != NULL; mdl = nextMdl) {
            nextMdl = mdl->Next;
            MmUnlockPages( mdl ); IoFreeMdl( mdl ); // This function will also unmap pages.
        }
        Irp->MdlAddress = NULL;
    }

    if(Context) {
        ExFreePool(Context);
    }



    //
    // If you intend to queue the IRP and reuse it for another request,
    // make sure you call IoReuseIrp(Irp, STATUS_SUCCESS) before you reuse.
    //
    IoFreeIrp(Irp);

    //
    // NOTE: this is the only status that you can return for driver-created asynchronous IRPs.
    //
    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Scenario 11: Send an asynchronous request by using IoAllocateIrp

This scenario is similar to the previous scenario except that instead of calling **IoBuildAsynchronousFsdRequest**, this scenario uses the **IoAllocateIrp** function to create the IRP.

```
NTSTATUS
MakeAsynchronousRequest2(
    PDEVICE_OBJECT   TopOfDeviceStack,
    PVOID               WriteBuffer,
    ULONG               NumBytes
    )
/*++
Arguments:

    TopOfDeviceStack -

    WriteBuffer       - Buffer to be sent to the TopOfDeviceStack.

    NumBytes  - Size of buffer to be sent to the TopOfDeviceStack.

--*/
{
    NTSTATUS        status;
    PIRP            irp;
    LARGE_INTEGER   startingOffset;
    KEVENT          event;
    PIO_STACK_LOCATION  nextStack;

    startingOffset.QuadPart = (LONGLONG) 0;

    //
    // Start by allocating the IRP for this request.  Do not charge quota
    // to the current process for this IRP.
    //
```

```
    irp = IoAllocateIrp( TopOfDeviceStack->StackSize, FALSE );
    if (NULL == irp) {

        return STATUS_INSUFFICIENT_RESOURCES;
    }

     //
    // Obtain a pointer to the stack location of the first driver that will be
    // invoked.  This is where the function codes and the parameters are set.
    //

    nextStack = IoGetNextIrpStackLocation( irp );
    nextStack->MajorFunction = IRP_MJ_WRITE;
    nextStack->Parameters.Write.Length = NumBytes;
    nextStack->Parameters.Write.ByteOffset= startingOffset;


    if(TopOfDeviceStack->Flags & DO_BUFFERED_IO) {

        irp->AssociatedIrp.SystemBuffer = WriteBuffer;
        irp->MdlAddress = NULL;

    } else if (TopOfDeviceStack->Flags & DO_DIRECT_IO) {
        //
        // The target device supports direct I/O operations.  Allocate
        // an MDL large enough to map the buffer and lock the pages into
        // memory.
        //
        irp->MdlAddress = IoAllocateMdl( WriteBuffer,
                                         NumBytes,
                                         FALSE,
                                         FALSE,
                                         (PIRP) NULL );
        if (irp->MdlAddress == NULL) {
            IoFreeIrp( irp );
            return STATUS_INSUFFICIENT_RESOURCES;
        }

        try {

            MmProbeAndLockPages( irp->MdlAddress,
                                 KernelMode,
                                 (LOCK_OPERATION) (nextStack->MajorFunction == IRP_MJ_WRITE ? IoReadAccess :
IoWriteAccess) );

        } except(EXCEPTION_EXECUTE_HANDLER) {

              if (irp->MdlAddress != NULL) {
                  IoFreeMdl( irp->MdlAddress );
              }
              IoFreeIrp( irp );
              return  GetExceptionCode();

        }
    }

    IoSetCompletionRoutine(irp,
                   MakeAsynchronousRequestCompletion2,
                   NULL,
                   TRUE,
                   TRUE,
                   TRUE);

    (void) IoCallDriver(TargetDeviceObject, irp);

    return STATUS_SUCCESS;
}

NTSTATUS
```

```
NTSTATUS
MakeAsynchronousRequestCompletion2(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    PMDL mdl, nextMdl;

    //
    // Free any associated MDL.
    //

    if (Irp->MdlAddress != NULL) {
        for (mdl = Irp->MdlAddress; mdl != NULL; mdl = nextMdl) {
            nextMdl = mdl->Next;
            MmUnlockPages( mdl ); IoFreeMdl( mdl ); // This function will also unmap pages.
        }
        Irp->MdlAddress = NULL;
    }

    //
    // If you intend to queue the IRP and reuse it for another request,
    // make sure you call IoReuseIrp(Irp, STATUS_SUCCESS) before you reuse.
    //

    IoFreeIrp(Irp);

    return STATUS_MORE_PROCESSING_REQUIRED;
}
```

Scenario 12: Send an asynchronous request and cancel it in a different thread

This scenario shows you how you can send one request at a time to a lower driver without waiting for the request to complete, and you can also cancel the request at any time from another thread.

You can remember the IRP and other variables to do this work in a device extension or in a context structure global to the device as shown below. The state of the IRP is tracked with an IRPLOCK variable in the device extension. The IrpEvent is used to make sure that the IRP is fully completed (or freed) before making the next request.

This event is also useful when you handle IRP_MN_REMOVE_DEVICE and IRP_MN_STOP_DEVICE PNP requests where you have to make sure that there are no pending IRPs before you complete these requests. This event works best when you initialize it as a synchronization event in AddDevice or in some other initialization routine.

```
typedef struct _DEVICE_EXTENSION{
    ..
    PDEVICE_OBJECT TopOfDeviceStack;
    PIRP PendingIrp;
    IRPLOCK IrpLock; // You need this to track the state of the IRP.
    KEVENT IrpEvent; // You need this to synchronize various threads.
    ..
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
                </pre><pre class="code">
InitializeDeviceExtension( PDEVICE_EXTENSION  DeviceExtension)
{
    KeInitializeEvent(&DeviceExtension->IrpEvent, SynchronizationEvent, TRUE);
}

NTSTATUS
MakeASynchronousRequest3(
    PDEVICE_EXTENSION  DeviceExtension,
    PVOID              WriteBuffer,
    ULONG              NumBytes
    )
/*++
Arguments:
```

```
    DeviceExtension -

    WriteBuffer       - Buffer to be sent to the TargetDeviceObject.

    NumBytes  - Size of buffer to be sent to the TargetDeviceObject.

--*/
{
    NTSTATUS        status;
    PIRP            irp;
    LARGE_INTEGER   startingOffset;
    PIO_STACK_LOCATION  nextStack;

    //
    // Wait on the event to make sure that PendingIrp
    // field is free to be used for the next request. If you do
    // call this function in the context of the user thread,
    // make sure to call KeEnterCriticialRegion before the wait to protect
    // the thread from getting suspended while holding a lock.
    //
    KeWaitForSingleObject( &DeviceExtension->IrpEvent,
                           Executive,
                           KernelMode,
                           FALSE,
                           NULL );

    startingOffset.QuadPart = (LONGLONG) 0;
    //
    // If the IRP is used for the same purpose every time, you can just create the IRP in the
    // Initialization routine one time and reuse it by calling IoReuseIrp.
    // The only thing that you have to do in the routines in this article
    // is remove the lines that call IoFreeIrp and set the PendingIrp
    // field to NULL. If you do so, make sure that you free the IRP
    // in the PNP remove handler.
    //
    irp = IoBuildAsynchronousFsdRequest(
                IRP_MJ_WRITE,
                DeviceExtension->TopOfDeviceStack,
                WriteBuffer,
                NumBytes,
                &startingOffset, // Optional
                NULL
                );

    if (NULL == irp) {

        return STATUS_INSUFFICIENT_RESOURCES;
    }

    //
    // Initialize the fields relevant fields in the DeviceExtension
    //
    DeviceExtension->PendingIrp = irp;
    DeviceExtension->IrpLock = IRPLOCK_CANCELABLE;

    IoSetCompletionRoutine(irp,
                   MakeASynchronousRequestCompletion3,
                   DeviceExtension,
                   TRUE,
                   TRUE,
                   TRUE);
    //
    // If you want to change any value in the IRP stack, you must
    // first obtain the stack location by calling IoGetNextIrpStackLocation.
    // This is the location that is initialized by the IoBuildxxx requests and
    // is the one that the target device driver is going to view.
    //
```

```
    nextStack = IoGetNextIrpStackLocation(irp);

    //
    // You could change the MajorFunction code to something appropriate.
    //
    nextStack->MajorFunction = IRP_MJ_SCSI;

    (void) IoCallDriver(DeviceExtension->TopOfDeviceStack, irp);

    return STATUS_SUCCESS;
}

NTSTATUS
MakeASynchronousRequestCompletion3(
    IN PDEVICE_OBJECT   DeviceObject,
    IN PIRP             Irp,
    IN PVOID            Context
    )
{
    PMDL mdl, nextMdl;
    PDEVICE_EXTENSION deviceExtension = Context;

    //
    // If the target device object is set up to do buffered i/o
    // (TargetDeviceObject->Flags & DO_BUFFERED_IO), then
    // IoBuildAsynchronousFsdRequest request allocates a system buffer
    // for read and write operation. If you stop the completion of the IRP
    // here, you must free that buffer.
    //

    if(Irp->AssociatedIrp.SystemBuffer && (Irp->Flags & IRP_DEALLOCATE_BUFFER) ) {
            ExFreePool(Irp->AssociatedIrp.SystemBuffer);
    }

    //
    // If the target device object is set up to do direct i/o (DO_DIRECT_IO), then
    // IoBuildAsynchronousFsdRequest creates an MDL to describe the buffer
    // and locks the pages. If you stop the completion of the IRP, you must unlock
    // the pages and free the MDL.
    //

    if (Irp->MdlAddress != NULL) {
        for (mdl = Irp->MdlAddress; mdl != NULL; mdl = nextMdl) {
            nextMdl = mdl->Next;
            MmUnlockPages( mdl ); IoFreeMdl( mdl ); // This function will also unmap pages.
        }
        Irp->MdlAddress = NULL;
    }

    if (InterlockedExchange((PVOID)&deviceExtension->IrpLock, IRPLOCK_COMPLETED)
                    == IRPLOCK_CANCEL_STARTED) {
        //
        // Main line code has got the control of the IRP. It will
        // now take the responsibility of freeing the IRP.
        // Therefore...
        return STATUS_MORE_PROCESSING_REQUIRED;
    }

    //
    // If you intend to queue the IRP and reuse it for another request, make
    // sure you call IoReuseIrp(Irp, STATUS_SUCCESS) before you reuse.
    //
    IoFreeIrp(Irp);
    deviceExtension->PendingIrp = NULL; // if freed
    //
    // Signal the event so that the next thread in the waiting list
    // can send the next request.
    //
    KeSetEvent (&deviceExtension->IrpEvent, IO_NO_INCREMENT, FALSE);
```

```
    return STATUS_MORE_PROCESSING_REQUIRED;
}

VOID
CancelPendingIrp(
    PDEVICE_EXTENSION DeviceExtension
    )
/*++
    This function tries to cancel the PendingIrp if it is not already completed.
    Note that the IRP may not be completed and freed when the
    function returns. Therefore, if you are calling this from your PNP Remove device handle,
    you must wait on the IrpEvent to make sure the IRP is indeed completed
    before successfully completing the remove request and allowing the driver to unload.
--*/
{
     if (InterlockedExchange((PVOID)&DeviceExtension->IrpLock, IRPLOCK_CANCEL_STARTED) == IRPLOCK_CANCELABLE) {

        //
        // You got it to the IRP before it was completed. You can cancel
        // the IRP without fear of losing it, as the completion routine
        // will not let go of the IRP until you say so.
        //
        IoCancelIrp(DeviceExtension->PendingIrp);
        //
        // Release the completion routine. If it already got there,
        // then you need to free it yourself. Otherwise, you got
        // through IoCancelIrp before the IRP completed entirely.
        //
        if (InterlockedExchange((PVOID)&DeviceExtension->IrpLock, IRPLOCK_CANCEL_COMPLETE) ==
IRPLOCK_COMPLETED) {
            IoFreeIrp(DeviceExtension->PendingIrp);
            DeviceExtension->PendingIrp = NULL;
            KeSetEvent(&DeviceExtension->IrpEvent, IO_NO_INCREMENT, FALSE);
        }

     }

    return ;
}
```

References

- Walter Oney. Programming Windows Driver Model, Second Edition, Chapter 5.

I/O Programming Techniques

6/25/2019 • 2 minutes to read • Edit Online

This section describes programming techniques that can be used to work with objects managed by the I/O manager. The following technology areas are discussed:

General I/O Programming Techniques

Methods for Accessing Data Buffers

DMA Programming Techniques

PIO Programming Techniques

Legacy I/O Programming

For architectural information on the I/O manager, see Windows I/O Manager. For reference information on I/O manager, see I/O Manager Routines.

General I/O Programming Techniques

12/5/2018 • 2 minutes to read • Edit Online

One of the most important techniques in I/O programming is one that you should avoid: forcing the operating system to wait for your device. Almost everyone has had the experience of seeing Microsoft Windows "freeze up". Sometimes the freeze is due to a crash, but other times the system is simply waiting for a device to respond.

There are two basic programming techniques for dealing with waiting for a device: *synchronous* and *asynchronous*. Synchronous programming waits for the device and should be avoided. Asynchronous programming uses other techniques (such as waiting for interrupt requests). For more information about synchronous and asynchronous programming, see the following topics:

Synchronous I/O Programming

Asynchronous I/O Programming

Microsoft Vista has a new policy for dealing with problems with synchronous programming. For more information about this new policy, see Restricting Waits in Windows Vista for more information.

In earlier device driver programming, a driver would need to repeatedly request information from a driver until the answer was provided. This technique is called polling and should almost never be used. The best way to handle the problem of polling is to use hardware interrupts. For more information about hardware interrupts, see Servicing Interrupts. For more information on polling and why you should not use it, see Avoid Device Polling.

Synchronous I/O Programming

12/5/2018 • 2 minutes to read • Edit Online

Synchronous programming simply waits for a call to return. This is fast and efficient from the programmer's point of view but in an environment like Windows where many programs are running at once, it can cause problems. Whenever possible, use Asynchronous I/O Programming Techniques.

Note For driver developers using Microsoft Vista, this is not as serious a problem. For more information about synchronous programming in Vista, see Restricting Waits in Vista.

Asynchronous I/O Programming

12/5/2018 • 2 minutes to read • Edit Online

Asynchronous programming does not force everyone else to wait. This is the preferred technique for programming Windows device drivers. Supporting asynchronous I/O is one of the design goals of WDM drivers. For more information about asynchronous I/O in drivers, see Supporting Asynchronous I/O. For device drivers, using interrupts is the best way to program asynchronously. You simply send a request to your device and let the system take control. Then when your device wants to tell you something, it triggers an interrupt that the operating system processes by calling an interrupt handler in your driver. This communication is handled through IRPs. For more information about IRPS, see Handling IRPs.

Restricting Waits in Vista

12/5/2018 • 2 minutes to read • Edit Online

Because many device driver developers use Synchronous I/O Programming Techniques, Windows can slow down or "freeze up" while a device is taking time to respond. To reduce this problem, the I/O Manager in Vista will stop execution of programs that are "stuck" waiting for a device to respond after a few moments.

Note It is strongly recommended that Synchronous I/O Programming Techniques are avoided in your device driver. If Vista stops execution of your driver code because your driver is waiting for a device, your device may be left in an unknown state.

Avoid Polling Devices

6/25/2019 • 2 minutes to read • Edit Online

A device driver should avoid polling its device unless absolutely necessary, and should never use a whole time-slice for polling. Polling a device is an expensive operation that makes any operating system compute-bound within the polling driver. A device driver that does much polling interferes with I/O operations on other devices and can make the system slow and unresponsive to users.

Recently developed devices, which are as technologically advanced as the processors on which Windows is designed to run, seldom require a driver to poll its device, either to ensure that the device is ready to start an I/O operation or that an operation is complete.

Nevertheless, some devices still in use were designed to work with old processors, which had narrow data buses, slow clock rates, and single-user, single-tasking operating systems that did synchronous I/O. Such devices might require polling or some other means of waiting for the device to update its registers.

Although it might seem logical to solve a slow-device problem by coding a simple loop that increments a counter, thereby "wasting" a minimum interval while the device updates registers, such a driver is unlikely to be portable across Windows platforms. The loop counter maximum would require customization for each platform. Furthermore, if the driver is compiled with a good optimizing compiler, the compiler might remove the driver's counter variable and the loop(s) where it is incremented.

Note Follow this implementation guideline if the driver must stall while the device hardware updates state: A driver can call **KeStallExecutionProcessor** before it reads device registers. The driver should minimize the interval it stalls and should, in general, specify a stall interval no longer than 50 microseconds.

The granularity of a **KeStallExecutionProcessor** interval is one microsecond.

If the device frequently requires more than 50 microseconds to update state, consider setting up a device-dedicated thread in the driver.

Maintaining Cache Coherency

12/5/2018 • 2 minutes to read • Edit Online

When a driver is transferring data between system memory and its device, data can be cached in one or more processor caches and/or in the system DMA controller's cache. Drivers that use DMA or PIO to service read/write IRPs or any device I/O control request that requires a DMA or PIO data transfer operation should ensure the integrity of possibly cached data during transfer operations. This section explains how to do so.

This section contains the following topics:

Methods for Accessing Data Buffers

6/25/2019 • 2 minutes to read • Edit Online

One of the primary responsibilities of driver stacks is transferring data between user-mode applications and a system's devices. The operating system provides the following three methods for accessing data buffers:

Buffered I/O
The operating system creates a nonpaged system buffer, equal in size to the application's buffer. For write operations, the I/O manager copies user data into the system buffer before calling the driver stack. For read operations, the I/O manager copies data from the system buffer into the application's buffer after the driver stack completes the requested operation.

For more information, see Using Buffered I/O.

Direct I/O
The operating system locks the application's buffer in memory. It then creates a memory descriptor list (MDL) that identifies the locked memory pages, and passes the MDL to the driver stack. Drivers access the locked pages through the MDL.

For more information, see Using Direct I/O.

Neither Buffered Nor Direct I/O
The operating system passes the application buffer's virtual starting address and size to the driver stack. The buffer is only accessible from drivers that execute in the application's thread context.

For more information, see Using Neither Buffered Nor Direct I/O.

For **IRP_MJ_READ** and **IRP_MJ_WRITE** requests, drivers specify the I/O method by using flags in each **DEVICE_OBJECT** structure. For more information, see Initializing a Device Object.

For **IRP_MJ_DEVICE_CONTROL** and **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests, the I/O method is determined by the *TransferType* value that is contained in each IOCTL value. For more information, see Defining I/O Control Codes.

All drivers in a driver stack must use the same buffer access method for each request, except possibly for the highest-level driver (which can use the "neither" method, regardless of the method used by lower drivers).

Using Buffered I/O

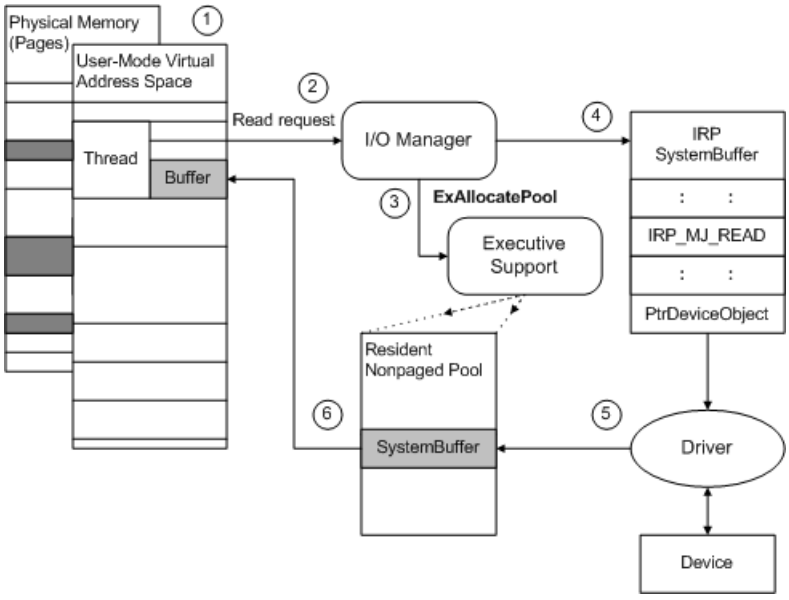
6/25/2019 • 3 minutes to read • Edit Online

A driver that services an interactive or slow device, or one that usually transfers relatively small amounts of data at a time, should use the buffered I/O transfer method. Using buffered I/O for small, interactive transfers improves overall physical memory usage, because the memory manager does not need to lock down a full physical page for each transfer, as it does for drivers that request direct I/O. Generally, video, keyboard, mouse, serial, and parallel drivers request buffered I/O.

The I/O manager determines that an I/O operation is using buffered I/O as follows:

- For **IRP_MJ_READ** and **IRP_MJ_WRITE** requests, DO_BUFFERED_IO is set in the **Flags** member of the **DEVICE_OBJECT** structure. For more information, see Initializing a Device Object.
- For **IRP_MJ_DEVICE_CONTROL** and **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests, the IOCTL code's value contains METHOD_BUFFERED as the *TransferType* value in the IOCTL value. For more information, see Defining I/O Control Codes.

The following figure illustrates how the I/O manager sets up an **IRP_MJ_READ** request for a transfer operation that uses buffered I/O.



The figure shows an overview of how drivers can use the **SystemBuffer** pointer in the IRP to transfer data for a read request, when a driver has ORed the device object's **Flags** with DO_BUFFERED_IO:

1. Some range of user-space virtual addresses represents the current thread's buffer, and that buffer's contents might be stored somewhere within a range of page-based physical addresses (dark shading in the previous figure).
2. The I/O manager services the current thread's read request, for which the thread passes a range of user-space virtual addresses representing a buffer.
3. The I/O manager checks the user-supplied buffer for accessibility and calls **ExAllocatePoolWithTag** to create a nonpaged system-space buffer (**SystemBuffer**) the size of the user-supplied buffer.
4. The I/O manager provides access to the newly allocated **SystemBuffer** in the IRP it sends to the driver.

   If the figure showed a write request, the I/O manager would copy data from the user buffer into the system

buffer before it sent the IRP to the driver.

5. For the read request shown in the previous figure, the driver reads data from the device into the system-space buffer. The memory for this buffer is nonpaged and the driver can safely access the buffer without first locking it. When the read request has been satisfied, the driver calls **IoCompleteRequest** with the IRP.

6. When the original thread is again active, the I/O manager copies the read-in data from the system buffer into the user buffer. It also calls **ExFreePool** to release the system buffer.

After the I/O manager has created a system-space buffer for the driver, the requesting user-mode thread can be swapped out and its physical memory can be reused by another thread, possibly by a thread belonging to another process. However, the system-space virtual address range supplied in the IRP remains valid until the driver calls **IoCompleteRequest** with the IRP.

Drivers that transfer large amounts of data at a time, in particular, drivers that do multipage transfers, should not attempt to use buffered I/O. As the system runs, nonpaged pool can become fragmented so that the I/O manager cannot allocate large, contiguous system-space buffers to send in IRPs for such a driver.

Typically, a driver uses buffered I/O for some types of IRPs, such as **IRP_MJ_DEVICE_CONTROL** requests, even if it also uses direct I/O. Drivers that use direct I/O typically only do so for **IRP_MJ_READ** and **IRP_MJ_WRITE** requests, and possibly driver-defined **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests that require large data transfers.

Every **IRP_MJ_DEVICE_CONTROL** and **IRP_MJ_INTERNAL_DEVICE_CONTROL** request includes an I/O control code. If the I/O control code indicates that the IRP must be supported by using buffered I/O, the I/O manager uses a single system buffer to represent the user application's input and output buffers. A driver that supports such an I/O control code must read input data (if any) from the buffer and then supply output data (if any) by overwriting the input data. For more information, see Defining I/O Control Codes.

Using Direct I/O

6/25/2019 • 2 minutes to read • Edit Online

Drivers for devices that can transfer large amounts of data at a time should use direct I/O for those transfers. Using direct I/O for large transfers improves a driver's performance, both by reducing its interrupt overhead and by eliminating the memory allocation and copying operations inherent in buffered I/O.

Generally, mass-storage device drivers request direct I/O for transfer requests, including lowest-level drivers that use direct memory access (DMA) or programmed I/O (PIO), as well as any intermediate drivers chained above them.

The I/O manager determines that an I/O operation is using direct I/O as follows:

- For **IRP_MJ_READ** and **IRP_MJ_WRITE** requests, DO_DIRECT_IO is set in the **Flags** member of the **DEVICE_OBJECT** structure. For more information, see Initializing a Device Object.
- For **IRP_MJ_DEVICE_CONTROL** and **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests, the IOCTL code's value contains METHOD_IN_DIRECT or METHOD_OUT_DIRECT as the *TransferType* value in the IOCTL value. For more information, see Defining I/O Control Codes.

Drivers that use direct I/O will sometimes also use buffered I/O to handle some IRPs. In particular, drivers typically use buffered I/O for some I/O control codes for **IRP_MJ_DEVICE_CONTROL** requests that require data transfers, regardless of whether the driver uses direct I/O for read and write operations.

Setting up a direct I/O transfer varies slightly, depending on whether DMA or PIO is being used. For more information, see:

Using Direct I/O with DMA

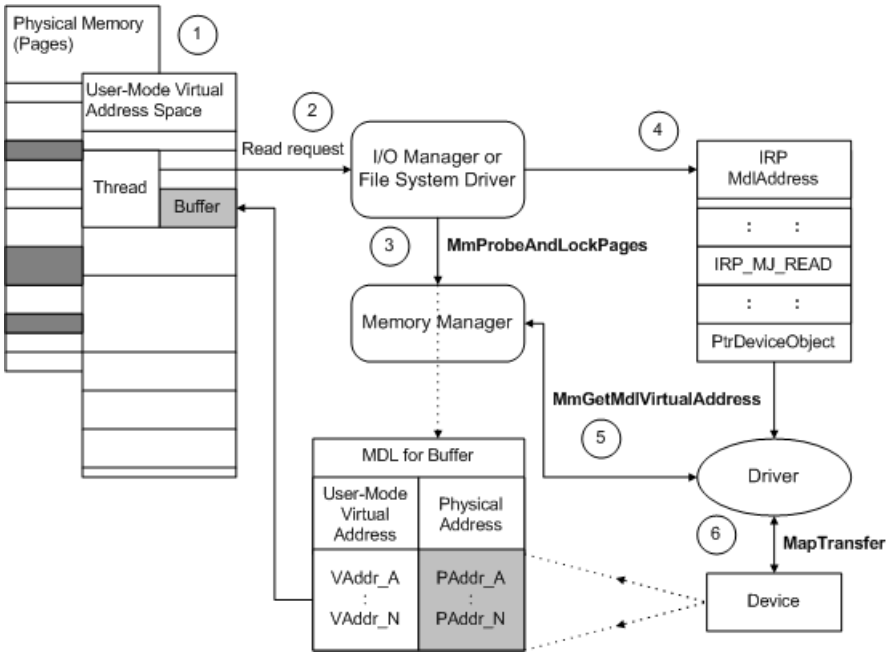
Using Direct I/O with PIO

Drivers must take steps to maintain cache coherency during DMA and PIO transfers. For more information, see Maintaining Cache Coherency.

Using Direct I/O with DMA

6/25/2019 • 2 minutes to read • Edit Online

The following figure illustrates how the I/O manager sets up an **IRP_MJ_READ** request for a DMA transfer operation that uses direct I/O.



The previous figure illustrates how drivers can use the IRP's **MdlAddress** to transfer data for a read request. The driver in the figure uses packet-based system or bus-master DMA, and has ORed the device object's **Flags** with DO_DIRECT_IO.

1. Some range of user-space virtual addresses represents the current thread's buffer, and that buffer's contents might actually be stored on some number of physically discontiguous pages (dark shading in the previous figure). The I/O manager creates an MDL to describe this buffer. An MDL is an opaque data structure, defined by the memory manager, that maps a particular virtual address range to one or more page-based physical address ranges. For more information, see Using MDLs.

2. The I/O manager services the current thread's read request, for which the thread passes a range of user-space virtual addresses that represent a buffer.

3. The I/O manager or file system driver (FSD) checks the user-supplied buffer for accessibility and calls **MmProbeAndLockPages** with the previously created MDL. **MmProbeAndLockPages** also fills in the corresponding physical address range in the MDL.

   As the previous figure shows, an MDL for a virtual range can have several corresponding page-based physical address entries, and the virtual range for a buffer might begin and end at some byte offset from the start of the first and last pages described by an MDL.

4. The I/O manager provides a pointer to the MDL (**MdlAddress**) in an IRP that requests a transfer operation. Until the I/O manager or file system calls **MmUnlockPages** after the driver completes the IRP, the physical pages described in the MDL remain locked down and assigned to the buffer. However, the virtual addresses in such an MDL can become invisible (and invalid), even before the IRP is sent to the device driver or to any intermediate driver that might be layered above the device driver.

5. If the driver uses packet-based system or bus-master DMA, its *AdapterControl* routine calls

MmGetMdlVirtualAddress with the IRP's **MdlAddress** pointer to get the base virtual address for the MDL's page-based entries.

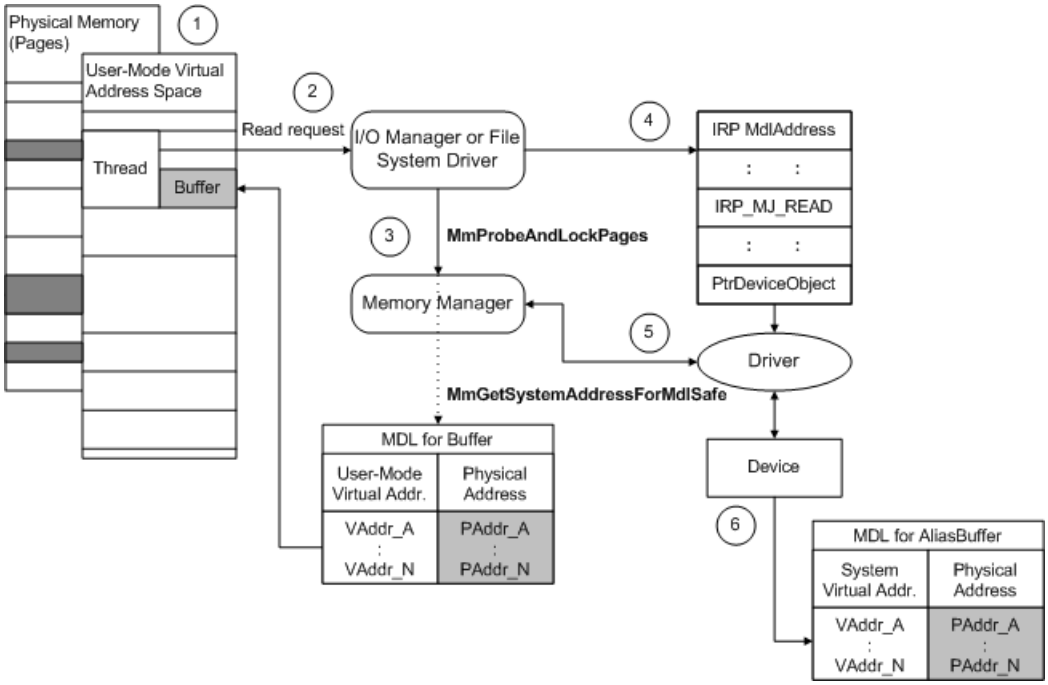
6. The *AdapterControl* routine then calls **MapTransfer** with the base address returned by **MmGetMdlVirtualAddress**, to read data from the device directly into physical memory. (For more information, see Adapter Objects and DMA.)

Drivers should always check buffer lengths. Note that the I/O manager does not create an MDL for a zero-length buffer.

Using Direct I/O with PIO

6/25/2019 • 2 minutes to read • Edit Online

A driver that uses programmed I/O (PIO) rather than DMA must doubly map user-space buffers into a system-space address range. The following figure illustrates how the I/O manager sets up an **IRP_MJ_READ** request for a PIO transfer operation that uses direct I/O.



The figure shows how a device that uses PIO handles the same task.

1. Some range of user-space virtual addresses represents the current thread's buffer, and that buffer's contents might actually be stored on some number of physically discontiguous pages. If the buffer length is nonzero, the I/O manager creates an MDL to describe this buffer.

2. The I/O manager services the current thread's read request, for which the thread passes a range of user-space virtual addresses representing a buffer.

3. The I/O manager or FSD checks the user-supplied buffer for accessibility. If the I/O manager has created an MDL, it calls **MmProbeAndLockPages** with an MDL, which specifies the range of virtual addresses for the user buffer. **MmProbeAndLockPages** also fills in the corresponding physical address range in the MDL.

4. The I/O manager provides a pointer to the MDL (**MdlAddress**) in an IRP that requests a transfer operation. Until the I/O manager or file system calls **MmUnlockPages** after the driver completes the IRP, the physical pages described in the MDL remain locked down and assigned to the buffer. However, the virtual addresses in such an MDL can become invisible (and invalid), even before the IRP is sent to the device driver or to any intermediate driver that might be layered above the device driver.

5. If the driver requires system (virtual) addresses, the driver calls **MmGetSystemAddressForMdlSafe** with the IRP's **MdlAddress** pointer to doubly map the user-space virtual addresses in the MDL to a system-space address range. In the figure above, AliasBuff represents the MDL that describes the doubly-mapped addresses.

6. The driver uses the system-space virtual address range from the doubly mapped MDL (AliasBuff) to read data into memory.

When the driver completes the IRP by calling **IoCompleteRequest**, the I/O manager or file system releases the MDL's doubly mapped system-space range if the driver called **MmGetSystemAddressForMdlSafe**. The I/O manager or file system unlocks the pages described in the MDL, and disposes of the MDL and IRP on the driver's behalf. For better performance, drivers should avoid doubly mapping MDL physical addresses to system space, as described in step 3, unless they must use virtual addresses. Doing so uses system page-table entries unnecessarily and can decrease both driver performance and scalability. In addition, the system might crash if it runs out of page-table entries, because most older drivers cannot handle this situation.

The current user thread's buffers and the thread itself are guaranteed to be resident in physical memory only while that thread is current. For the thread shown in the previous figure, its user buffer's contents could be paged out to secondary storage while another process's threads are run. When another process's thread is run, the system physical memory for the requesting thread's buffer can be overwritten unless the memory manager has locked down and preserved the corresponding physical pages that contain the original thread's buffer.

However, the original thread's virtual addresses for its buffer do not remain visible while another thread is current, even if the memory manager preserves the buffer's physical pages. Consequently, drivers cannot use a virtual address returned by **MmGetMdlVirtualAddress** to access memory. Callers of this routine must pass its results to **MapTransfer** (along with the IRP's **MdlAddress** pointer) in order to transfer data using packet-based system or bus-master DMA.

Using Neither Buffered Nor Direct I/O

6/25/2019 • 2 minutes to read • Edit Online

If a driver is using neither buffered nor direct I/O, then the I/O manager passes the original user-space virtual addresses in IRPs that it sends to the driver. To access these buffers safely, the driver must be executing in the context of the calling thread. Typically, therefore, only highest-level drivers, such as FSDs, can use this method for accessing buffers.

An intermediate or lowest-level driver cannot always meet this condition. For example, if a requesting thread waits on the completion of an I/O request or if a higher-level driver is layered over the intermediate or lowest-level driver, then the lower-level driver's routines are unlikely to be called in the context of the requesting thread.

The I/O manager determines that an I/O operation is using neither buffered nor direct I/O as follows:

- For **IRP_MJ_READ** and **IRP_MJ_WRITE** requests, neither DO_BUFFERED_IO nor DO_DIRECT_IO are set in the **Flags** member of the **DEVICE_OBJECT** structure. For more information, see Initializing a Device Object.
- For **IRP_MJ_DEVICE_CONTROL** and **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests, the IOCTL code's value contains METHOD_NEITHER as the *TransferType* value in the IOCTL value. For more information, see Defining I/O Control Codes.

When a driver receives an IRP that specifies an I/O operation using neither buffered nor direct I/O, it must do the following:

1. Check the validity of the user buffer's address range and check whether the appropriate read or write access is permitted, using the **ProbeForRead** and **ProbeForWrite** support routines. The driver must enclose its accesses to the buffer's address range within a driver-supplied exception handler, so that a user thread cannot change the access rights for the buffer while the driver is accessing memory. If the probe raises an exception, the driver should return an error. The driver must call these routines within the context of the thread that made the I/O request; therefore, only a higher-level driver can perform this task.
2. Manage buffers and memory operations in one of the following ways:
   - Carry out its own double-buffering operations, as the I/O manager does for drivers that use buffered I/O. For more information, see Using Buffered I/O.
   - Create its own MDLs and lock down the buffer by calling the memory manager's support routines, as the I/O manager does for drivers that use direct I/O. For more information, see Using Direct I/O.
   - Perform all necessary operations on the user buffer directly in the context of the calling thread. The driver must wrap its access to the buffer within a driver-supplied exception handler, in case a user thread changes either the access rights for the buffer or the data in the buffer while the driver is accessing memory. For more information, see Handling Exceptions.

In effect, the driver must choose on a per-IRP basis whether to do buffered I/O, direct I/O, or I/O in the context of the calling thread, and it must handle any exceptions that might occur in a user-mode thread context. The driver must manage its own user buffer accesses, double-buffering operations, and memory mappings, as necessary, instead of letting the I/O manager handle these operations for the driver.

DMA Programming Techniques

12/5/2018 • 2 minutes to read • Edit Online

Direct Memory Access (DMA) is one of the most basic hardware techniques for transferring memory-based data between the central processor (CPU) and a particular device. Computer systems use a DMA controller which is an intermediate device that handles the memory transfer, allowing the CPU to do other things.

Drivers can use the DMA controller to transfer memory-based data directly. The following topics discuss DMA issues related to I/O programming.

Drivers can use adapter objects to control DMA. For more information about adapter objects, see Adapter Objects and DMA.

When a driver is transferring data between system memory and its device, data can be cached in one or more processor caches and/or in the system DMA controller's cache. For more information about DMA and caches, see Flushing Cached Data during DMA Operations.

If you need to split up your DMA operations into smaller chunks, see Splitting DMA Transfer Requests.

Version 3 of the DMA operations interface is available starting with Windows 8. For more information about this interface, see Version 3 of the DMA Operations Interface.

Flushing Cached Data during DMA Operations

6/25/2019 • 2 minutes to read • Edit Online

In some platforms, the processor and system DMA controller (or bus-master DMA adapters) exhibit cache coherency anomalies. The following guidelines enable drivers that use version 1 or 2 of the DMA operations interface (see **DMA_OPERATIONS**) to maintain coherent cache states across all supported processor architectures, including architectures that do not contain hardware to automatically enforce cache coherency.

Note The guidelines in this topic apply only to drivers that use versions 1 and 2 of the DMA operations interface. Drivers that use version 3 of this interface must follow a different set of guidelines. For more information, see Version 3 of the DMA Operations Interface.

To maintain data integrity during DMA operations, lowest-level drivers must follow these guidelines

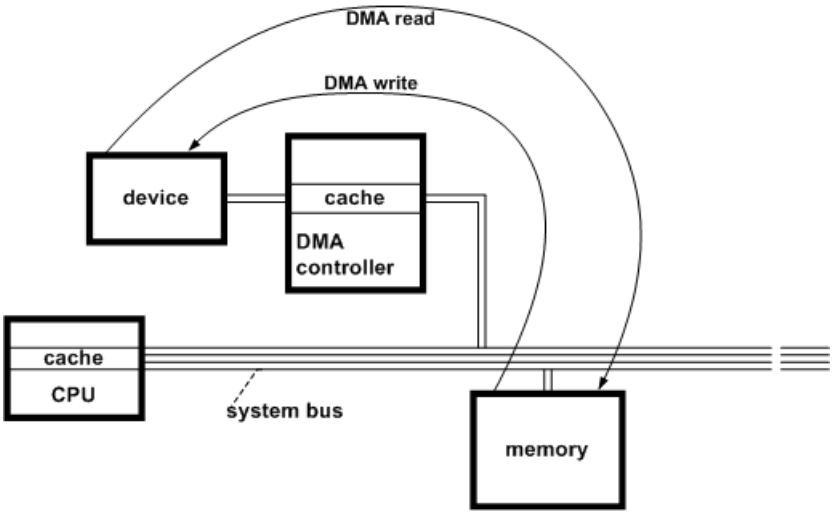
1. Call **KeFlushIoBuffers** before beginning a transfer operation to maintain consistency between data that might be cached in the processor and the data in memory.

   If a driver calls **AllocateCommonBuffer** with the *CacheEnabled* parameter set to **TRUE**, the driver must call **KeFlushIoBuffers** before beginning a transfer operation to/from its buffer.

2. Call **FlushAdapterBuffers** at the end of each device transfer operation to be sure any remainder bytes in the system DMA controller's buffers have been written into memory or to the subordinate device.

   Or, call **FlushAdapterBuffers** at the end of each transfer operation for a given IRP to be sure all data has been read into system memory or written out to a bus-master DMA device.

The following figure shows why it is important to flush the processor cache before a read or write operation using DMA if the host processor and DMA controller do not automatically maintain cache coherency.



An asynchronous DMA read or write operation accesses data in memory, not in the processor cache. Unless this cache has been flushed by calling **KeFlushIoBuffers** just before a read, the data transferred into system memory by the DMA operation could be overwritten with stale data if the processor cache is flushed later. Unless the processor cache has been flushed by calling **KeFlushIoBuffers** just before a write, the data in this cache might be more up-to-date than the copy in memory.

KeFlushIoBuffers does nothing if the processor and DMA controller can be relied on to maintain cache coherency, so calls to this support routine have almost no overhead in such a platform.

As also shown in the previous figure, DMA controllers, which are represented by adapter objects, can have internal

buffers. Such a DMA controller can transfer cached data in fixed-size chunks, usually eight or more bytes at a time. Moreover, these DMA controllers can wait until their internal buffers are full before each transfer operation.

Consider the case of a lowest-level driver that uses subordinate DMA to read data in variable-sized chunks or in fixed-size chunks that are not an integral multiple of a system DMA controller's cache size. Unless this driver calls **FlushAdapterBuffers** at the end of each device transfer, it cannot be sure when every byte the driver requested actually will be transferred.

The driver of a bus-master DMA device also should call **FlushAdapterBuffers** at the end of each transfer operation for an IRP to be sure that all data has been transferred into system memory or out to the device.

FlushAdapterBuffers returns a Boolean, value that indicates whether the requested flush operation was successful. A driver can use this value to determine how to set the I/O status block when completing an IRP for a DMA read or write operation.

Splitting DMA Transfer Requests

6/25/2019 • 3 minutes to read • Edit Online

Any driver might need to split up a transfer request and carry out more than one DMA transfer operation to satisfy a given IRP, depending on the following:

- The number of map registers returned by **IoGetDmaAdapter**
- The bytes of data to be transferred, contained in the **Length** member of the driver's I/O stack location for the IRP
- The number of page boundaries, in system physical memory, for the buffer into which or from which the driver is to transfer data
- Device-specific constraints on the driver's DMA operations. For example, the system "AT" disk driver must split up transfer requests for more than 256 sectors due to the disk controller's limitations.

A driver can determine the number of map registers needed to transfer all the data specified by an IRP as follows:

1. Call **MmGetMdlVirtualAddress**, passing a pointer to the MDL at **Irp->MdlAddress**, to get the starting virtual address for the buffer. Note that a driver must not attempt to access memory using this virtual address. The value returned by **MmGetMdlVirtualAddress** is an index into the MDL, not necessarily a valid address.
2. Pass the returned index and the value of **Length** in the driver's I/O stack location of the IRP to the **ADDRESS_AND_SIZE_TO_SPAN_PAGES** macro.

If the value returned by **ADDRESS_AND_SIZE_TO_SPAN_PAGES** is greater than the *NumberOfMapRegisters* value returned by **IoGetDmaAdapter**, the driver cannot transfer all requested data for this IRP in a single DMA operation. Instead, it must do the following:

1. Split the buffer into pieces that are sized to suit the number of available map registers (and any device-specific DMA constraints).
2. Carry out as many DMA operations as it takes to satisfy the transfer request.

For example, suppose **ADDRESS_AND_SIZE_TO_SPAN_PAGES** indicates that twelve map registers are needed to satisfy a transfer request, but the *NumberOfMapRegisters* value returned by **IoGetDmaAdapter** is only five. (Assume no device-specific DMA constraints.) In this case, the driver must carry out three DMA transfer operations, calling **MapTransfer** three times to transfer all the data requested by the IRP.

The system's DMA device drivers use various techniques to split up a DMA transfer when there are not enough map registers to satisfy an IRP with a single I/O operation. One technique to use is the following:

1. Call **IoAllocateMdl** to allocate an MDL describing a portion of the user buffer.
2. Call **MmProbeAndLockPages** to lock down that portion of the user buffer.
3. Transfer the data for that portion of the buffer.
4. Call **MmUnlockPages** and do either of the following:
   - If the MDL that the driver allocated in step 1 is large enough for the next piece of the transfer, call **MmPrepareMdlForReuse** and repeat steps 2 through 4.
   - Otherwise, call **IoFreeMdl** and repeat steps 1 through 4.

5. Call **MmUnlockPages** and **IoFreeMdl** when all the data has been transferred.

If a highest-level driver cannot lock down the entire user buffer with **MmProbeAndLockPages** in a machine with limited memory, it can do the following:

1. Call **IoBuildSynchronousFsdRequest** to allocate a partial-transfer IRP and lock down a portion of the user buffer. The locked-down area is usually either a multiple of **PAGE_SIZE** or is sized to suit the underlying device's transfer capacity.
2. Call **IoCallDriver** for the partial-transfer IRP, and call **KeWaitForSingleObject** to wait for an event object that the driver set up to be associated with its partial-transfer IRP, if lower drivers return STATUS_PENDING.
3. When it regains control, repeat steps 1 and 2 until all the data has been transferred, and, then, complete the original IRP.

When a storage class driver splits up large transfer requests for underlying SCSI port/miniport drivers, it allocates an additional IRP for each piece of the transfer request. It registers an *IoCompletion* routine for each driver-allocated IRP, to track the status of the full transfer request and to free the driver-allocated IRPs. Then it sends these IRPs on to the port driver using **IoCallDriver**.

Other class/port drivers can use this technique only if the class driver can determine how many map registers are available to the port driver. The port driver must store this configuration information in the registry for the paired class driver, or the paired drivers must define a private interface, using internal device I/O control requests, to pass configuration information about the number of available map registers from the port driver to the class driver.

A monolithic driver (that is, a driver not part of a class/port pair) for a DMA device must split up large transfer requests for itself. Such drivers usually split a large request into pieces and carry out a sequence of DMA operations in order to satisfy the IRP.

If a transfer request is too large for the underlying device driver to handle, a higher-level driver can call **MmGetMdlVirtualAddress** and **IoBuildPartialMdl**, then set up a sequence of partial-transfer IRPs for underlying device drivers.

Introduction to Adapter Objects

6/25/2019 • 2 minutes to read • Edit Online

Any driver that uses direct I/O and DMA must create an adapter object. The adapter object represents either a DMA controller channel or port, or a bus-master device.

Two kinds of lowest-level drivers must use adapter objects:

- Drivers for devices that use the system DMA controller. Such devices are called *subordinate devices* and are said to "use system (or *subordinate*) DMA."
- Drivers for devices that are bus-master adapters. Such devices arbitrate with the system for use of the I/O bus, and thus use bus-master DMA.

Drivers provide storage, usually in a device extension, for a pointer to the adapter object.

To carry out DMA transfers, drivers of devices that use either of these DMA methods usually have an *AdapterControl* routine and call system-supplied support routines that manipulate adapter objects. (Drivers that do not require *AdapterControl* routines include those that use scatter/gather DMA and those that use common-buffer, bus-master DMA.)

As part of device start-up operations, drivers that handle DMA operations call the I/O manager, which in turn calls the platform-specific HAL to create a set of adapter objects. On any Windows platform, the set of adapter objects usually includes an adapter object for:

- Each system DMA controller channel or port to which a subordinate device is attached.
- Each bus-master DMA device in the machine.

(For SCSI devices capable of bus-master DMA, the SCSI port driver sets up adapter objects for HBA-specific SCSI miniport drivers. The miniport driver's *HwScsiFindAdapter* routine supplies the port driver with adapter-specific data.)

See Using System DMA and Using Bus-Master DMA for more information about when and how drivers use adapter objects and *AdapterControl* routines.

Getting an Adapter Object

6/25/2019 • 3 minutes to read • Edit Online

At device start-up, a driver that uses system or bus-master DMA calls **IoGetDmaAdapter** to get a pointer to an adapter object and to determine the maximum number of map registers available for each transfer operation. When a driver calls **IoGetDmaAdapter**, the I/O manager, in turn, calls the HAL to get the necessary platform-specific information.

A driver must supply certain information in a system-defined **DEVICE_DESCRIPTION** structure in its call to **IoGetDmaAdapter**. Drivers must use **RtlZeroMemory** to initialize the **DEVICE_DESCRIPTION** structure with zeros before setting values in it.

The required data includes information about the features of the driver's device, such as whether the device is a bus master, if it has scatter/gather capabilities, and how many bytes of data the device can transfer at a time (**MaximumLength**).

The required device description data also includes platform-specific information, such as the platform-specific and system-assigned number of the bus that a driver of a bus-master device controls. A driver can obtain this information by calling **IoGetDeviceProperty**.

The **DEVICE_DESCRIPTION** structure includes some fields that might be irrelevant to some DMA devices or drivers. For example, the **BusNumber** field is not used in WDM drivers. Each driver should supply values for the relevant structure members and should set the values for all other members to zero.

The driver of a subordinate device should not pass **TRUE** in the **ScatterGather** field unless the device is capable of waiting for the system DMA controller to be reprogrammed when a request must be broken up into two or more DMA operations.

IoGetDmaAdapter returns both a pointer to an adapter object and a platform-specific or device-specific value indicating how many map registers are available with the adapter object for each DMA transfer operation.

The returned adapter object contains three fields that are accessible to drivers:

- Version number (**Version**)
- Size (**Size**)
- Pointer to a **DMA_OPERATIONS** structure (**DmaOperations**)

The **DMA_OPERATIONS** structure comprises a table of pointers to functions the driver must use to perform DMA operations on its device. The functions are accessible only through the pointers in this data structure; a driver cannot call them directly by name. (Note that these routines replace **Hal*Xxx*** routines supported in previous versions of Windows NT. To ensure compatibility for legacy drivers, the Wdm.h and Ntddk.h header files supply macros with the obsolete names, but new drivers should always call the functions through the data structure.)

The number of map registers can vary from device to device and from platform to platform. Generally, the HAL assigns a number of map registers according to the following criteria:

- If possible, the HAL returns a value that is one more than the number of map registers needed to transfer **MaximumLength** bytes, as specified in the driver's call to **IoGetDmaAdapter**.
- Otherwise, the HAL returns a lesser value that is as large as possible for the particular platform.

In other words, the HAL usually gives each driver enough map registers to maximize DMA throughput for its device, but the HAL can return a lesser value on some Windows platforms. There is no guarantee that a driver will

get the number of map registers it requests, so drivers should always check the returned value.

Any DMA device driver must provide storage for the adapter object pointer and *NumberOfMapRegisters* value returned by **IoGetDmaAdapter**. This pointer is a required parameter to the system-supplied support routines used for DMA. Because many of these support routines must be called at IRQL = DISPATCH_LEVEL, the driver-allocated storage must be resident. Most DMA drivers provide the necessary storage in a device extension. However, the storage can be in a controller extension if the driver also uses a controller object or in nonpaged pool that is allocated by the driver. See Allocating System-Space Memory and Managing Hardware Priorities for more information.

When the driver has completed all DMA operations, it calls **PutDmaAdapter** to free the adapter object.

The following sections (Using System DMA and Using Bus-Master DMA) describe how monolithic drivers of DMA devices use support routines to satisfy transfer requests. These sections assume that the driver has the following:

- A standard *StartIo* routine, rather than setting up and managing an internal queue of IRPs
- An internal routine to split transfer requests for which an insufficient number of map registers is available
- No device-specific DMA constraints

In other words, these sections describe the simplest possible technique for drivers' DMA operations, but individual drivers do not necessarily use exactly the same techniques. For any driver of a DMA device, which driver routines should split up large DMA transfer requests depends on the driver model (class/port or monolithic), on the device's features, and on any device-specific DMA constraints that driver must handle.

Using Scatter/Gather DMA

6/25/2019 • 3 minutes to read • Edit Online

Drivers that perform system or bus-master, packet-based DMA can use support routines designed especially for scatter/gather DMA. Instead of calling the sequence of routines outlined in Using Packet-Based System DMA and Packet-Based Bus-Master DMA, a driver can use **GetScatterGatherList** and **PutScatterGatherList**.

A device does not need to have built-in scatter/gather support for its driver to use these routines.

Drivers that use packet-based DMA call the following general sequence of support routines for scatter/gather operations:

1. **MmGetMdlVirtualAddress** to get an index into the MDL, required as a parameter in the call to **GetScatterGatherList**
2. **GetScatterGatherList** when the driver is ready to program its device for DMA and needs the system DMA controller or bus-master adapter

   GetScatterGatherList allocates the system DMA controller or bus-master adapter, determines how many map registers are required and allocates them, fills in the scatter/gather list, and calls the driver's *AdapterListControl* routine when the DMA controller or adapter and map registers are available.
3. **PutScatterGatherList** as soon as all the requested data has been transferred or the driver fails the IRP because of a device I/O error

   PutScatterGatherList flushes the adapter buffers, frees the map registers, and frees the scatter/gather list. The driver must call **PutScatterGatherList** before it can access the data in the buffer.

The adapter object pointer returned by **IoGetDmaAdapter** is a required parameter to each of these routines except **MmGetMdlVirtualAddress**, which requires a pointer to the MDL at *Irp*->**MdlAddress**.

The **GetScatterGatherList** routine includes calls to **AllocateAdapterChannel** and **MapTransfer**, so the driver does not have to make these calls. The routine takes the following as parameters:

- A pointer to the **DMA_ADAPTER** structure returned by **IoGetDmaAdapter**
- A pointer to the target device object for the DMA operation
- A pointer to the MDL that describes the buffer at *Irp*->**MdlAddress**
- A pointer to the current virtual address in the buffer described by the Mdl
- The number of bytes to be mapped
- A pointer to an *AdapterListControl* routine that performs the transfer
- A pointer to a driver-defined context area to be passed to the *AdapterListControl* routine
- A Boolean value: **TRUE** for a transfer to the device; **FALSE** otherwise

After determining the number of map registers required, allocating the adapter channel and map registers, filling in the scatter/gather list and preparing for the transfer, **GetScatterGatherList** calls the driver-supplied *AdapterListControl* routine. The *AdapterListControl* routine is run in an arbitrary thread context at IRQL = DISPATCH_LEVEL.

The *AdapterListControl* routine a driver supplies in calls to **GetScatterGatherList** differs from the *AdapterControl* routine passed to **AllocateAdapterChannel** in the following important respects:

- The *AdapterListControl* routine has no return value, whereas the *AdapterControl* routine returns an **IO_ALLOCATION_ACTION**.
- Rather than a pointer to the *MapRegisterBase* for the system-allocated map registers, the third parameter to an *AdapterListControl* routine instead points to a **SCATTER_GATHER_LIST** structure through which the driver can perform DMA.
- The *AdapterListControl* routine performs a subset of the tasks required in an *AdapterControl* routine.

  The *AdapterListControl* routine does not call **AllocateAdapterChannel** or **MapTransfer**. Its only responsibilities are to save the input scatter/gather list pointer, set up its device, and use the scatter/gather list to perform DMA.

The scatter/gather list structure includes a **SCATTER_GATHER_ELEMENT** array and the number of elements in the array. Each element of the array provides the length and starting physical address of a physically contiguous scatter/gather region. A driver uses the length and address in data transfers.

A driver can use **GetScatterGatherList** regardless of whether its device supports scatter/gather DMA. For a device that does not support scatter/gather DMA, the scatter/gather list will contain only one element.

Using the scatter/gather routines can improve performance over calling **AllocateAdapterChannel** (as previously described in Using Packet-Based System DMA and Using Packet-Based Bus-Master DMA). Unlike calls to **AllocateAdapterChannel**, more than one call to **GetScatterGatherList** can be queued for a device object at any one time. A driver can call **GetScatterGatherList** again for another DMA operation on the same driver object before its *AdapterListControl* routine has completed execution.

On return from the driver-supplied *AdapterListControl* routine, **GetScatterGatherList** keeps the map registers but frees the DMA adapter structure.

When the driver has satisfied the current IRP's transfer request or must fail the IRP due to a device or bus I/O error, it must call **PutScatterGatherList** before it can access the transferred data in the buffer. **PutScatterGatherList** flushes the adapter buffers and frees the map registers and scatter/gather list.

Writing AdapterControl Routines

6/25/2019 • 2 minutes to read • Edit Online

Most drivers of DMA devices have an *AdapterControl* routine, which is responsible for initiating DMA operations. (Drivers that do not require *AdapterControl* routines include those that use scatter/gather DMA and those that use common-buffer, bus-master DMA.)

When a driver calls **AllocateAdapterChannel**, its *AdapterControl* routine is run immediately if the system DMA controller or bus-master adapter is available for a DMA operation, and if enough map registers are available. Otherwise, the *AdapterControl* routine is queued until these resources are available.

If the driver's *AdapterControl* routine returns **KeepObject** or **DeallocateObjectKeepRegisters** (thereby retaining the system DMA controller channel or bus-master adapter for additional transfer operations), the driver's *DpcForIsr* or *CustomDpc* routine is responsible for releasing the adapter object or map registers by calling **FreeAdapterChannel** or **FreeMapRegisters** before the DPC routine completes the current IRP and returns control.

Storage Requirements for AdapterControl Routines

6/25/2019 • 2 minutes to read • Edit Online

If it has an *AdapterControl* routine, a driver must provide resident storage for the following:

- Context information to be used in its DMA operations
- An adapter object pointer returned by **IoGetDmaAdapter**
- A ULONG-type variable to hold the system-determined maximum *NumberOfMapRegisters* available for any given DMA transfer request

The driver can provide the necessary storage in a device extension, in a controller extension, or in nonpaged pool allocated by the driver.

Setting Up AdapterControl Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's dispatch routine for a PnP **IRP_MN_START_DEVICE** request must do the following for an *AdapterControl* routine:

1. Set up the adapter object for the device's DMA capabilities by filling in a **DEVICE_DESCRIPTION** structure and calling **IoGetDmaAdapter**.
2. Save the adapter object pointer and *NumberOfMapRegisters* returned by **IoGetDmaAdapter**.

   The platform-specific maximum *NumberOfMapRegisters* returned by **IoGetDmaAdapter** or the transfer capabilities of the driver's device, whichever is more restrictive, determines whether the driver must split up a given transfer request and carry out more than one DMA operation on its device to satisfy that IRP.

The returned adapter object pointer, the entry point of the driver's *AdapterControl* routine, the *DeviceObject* pointer representing the target device for the current IRP, a *Context* pointer to an area already set up for the *AdapterControl* routine, and a *NumberOfMapRegisters* value, which can be less than the maximum possible number for smaller transfer requests, must be passed in calls to **AllocateAdapterChannel**. Usually, a driver's *StartIo* (or possibly *ControllerControl*) routine sets up the area at *Context* before it calls **AllocateAdapterChannel**.

AdapterControl Routine Requirements

6/25/2019 • 2 minutes to read • Edit Online

At a minimum, an *AdapterControl* routine must do the following:

1. Save the input *MapRegisterBase* value along with any other context information that the driver needs to carry out one or more DMA transfer operations for the current IRP. The driver must pass the *MapRegisterBase* value to **FlushAdapterBuffers** when each DMA transfer operation is complete.
2. Return the appropriate **IO_ALLOCATION_ACTION** value:
   - **KeepObject** if the device is a subordinate device so the driver uses system DMA.
   - **DeallocateObjectKeepRegisters** if the device is a bus master so the driver uses packet-based, bus-master DMA.

Depending on the driver's design, its *AdapterControl* routine also can do the following before it returns control:

1. Determine the starting location for the transfer on its device.
2. Calculate the size of the transfer possible, given any limitations of its device due to the starting location of the transfer.

   In general, it is the responsibility of the routine that calls **AllocateAdapterChannel** to determine whether a transfer request must be split up into partial transfers due to any platform-specific limitations on the *NumberOfMapRegisters* available for each DMA transfer operation, as mentioned in the preceding section and detailed in Splitting Transfer Requests.
3. Set up any driver-maintained state about each transfer request in the device (or controller) extension.

   For example, an *AdapterControl* routine might call **KeSetTimer** with the entry point for a *CustomTimerDpc* routine that times out DMA transfer operations for the driver.
4. Call **MmGetMdlVirtualAddress** with the MDL pointer passed at **Irp->MdlAddress** to get an index for the start of the transfer, suitable for passing to **MapTransfer**.
5. Call **MapTransfer** to set up the system DMA controller or to obtain a physical-to-logical address mapping for a bus-master device.
6. Program the driver's device for a transfer operation, by using a *SynchCritSection* routine that is invoked by calling **KeSynchronizeExecution**. For more information, see Using Critical Sections.

If a transfer request requires the driver to perform a sequence of partial-transfer operations to satisfy the current IRP, the driver's *DpcForIsr* or *CustomDpc* routine is typically responsible for reprogramming the device for subsequent transfer operations. An *AdapterControl* routine is called only once for each incoming transfer IRP.

The driver routine that completes the current transfer IRP, usually the *DpcForIsr* or *CustomDpc* routine, also is responsible for releasing the system DMA controller or bus-master adapter by calling **FreeAdapterChannel** or **FreeMapRegisters**, respectively. This driver routine should make the appropriate call as soon as possible when its last partial-transfer operation is done so that drivers of subordinate DMA devices can allocate the system DMA controller or a bus-master driver can begin processing the next transfer IRP promptly.

Using Packet-Based System DMA

6/25/2019 • 2 minutes to read • Edit Online

Drivers of subordinate devices that use packet-based DMA call the following general sequence of support routines as they process an IRP requesting a DMA transfer:

- **KeFlushIoBuffers** just before attempting to allocate the system DMA controller (for more information, see Maintaining Cache Coherency)
- **AllocateAdapterChannel** when the driver is ready to program its device for DMA and needs the system DMA controller

  AllocateAdapterChannel, in turn, calls the driver's *AdapterControl* routine.
- **MmGetMdlVirtualAddress** to get an index into the MDL, required as an parameter in the initial call to **MapTransfer**
- **MapTransfer** to program the system DMA controller for the transfer operation

  A driver might need to call **MapTransfer** more than once to transfer all the requested data, as explained in Splitting Transfer Requests.
- **FlushAdapterBuffers** just after each DMA transfer operation to/from the subordinate device

  If a driver must call **MapTransfer** more than once to transfer all the requested data, it must call **FlushAdapterBuffers** as many times as it calls **MapTransfer**.
- **FreeAdapterChannel** either as soon as all the requested data has been transferred or if the driver fails the IRP because of a device I/O error

The adapter object pointer returned by **IoGetDmaAdapter** is a required parameter to each of these routines except **KeFlushIoBuffers** and **MmGetMdlVirtualAddress**, which require the pointer to the MDL passed at **Irp->MdlAddress**.

Individual drivers call this sequence of support routines at different points, depending on how each driver is implemented to service its device. For example, one driver's *StartIo* routine might make the call to **AllocateAdapterChannel**, another driver might make this call from a routine that removes IRPs from a driver-created interlocked queue, and still another driver might make this call when its subordinate DMA device indicates it is ready to transfer data.

Allocating an Adapter Channel for Packet-Based DMA

6/25/2019 • 3 minutes to read • Edit Online

To prepare for packet-based system DMA, a driver calls **KeFlushIoBuffers** and **AllocateAdapterChannel** after receiving an **IRP_MJ_READ** or **IRP_MJ_WRITE** request.

Before the driver calls these routines, its *DispatchRead* or *DispatchWrite* routine (or any other dispatch routine that handles a DMA transfer) should already have checked the validity of the IRP's parameters. The dispatch routine might also have queued the IRP to another driver routine for further processing.

The driver routine that calls **AllocateAdapterChannel** must be executing at IRQL = DISPATCH_LEVEL. Along with a pointer to the adapter object returned by **IoGetDmaAdapter**, a driver must supply the following when it calls **AllocateAdapterChannel**:

- A pointer to the target device object
- The entry point for its *AdapterControl* routine
- A pointer to any driver-determined context information the *AdapterControl* routine will use

AllocateAdapterChannel queues the driver's *AdapterControl* routine, which runs when the system DMA controller is assigned to this driver and a set of map registers has been allocated for the driver's DMA operation(s).

On entry, the *AdapterControl* routine receives the *DeviceObject* and *Context* pointers passed in the call to **AllocateAdapterChannel**, as well as a handle (*MapRegisterBase*) for the allocated map registers.

The *AdapterControl* routine also receives a pointer to the **DeviceObject->CurrentIrp** if the driver has a *StartIo* routine. If the driver manages its own queuing of IRPs (instead of having a *StartIo* routine), the driver should include a pointer to the current IRP as part of the context it passes when it calls **AllocateAdapterChannel**.

The *AdapterControl* routine typically does the following:

1. Saves or initializes whatever context the driver maintains about DMA operations. The context might include the input *MapRegisterBase* handle the driver must pass to **MapTransfer** and **FlushAdapterBuffers** and, possibly, the **Length** of the requested transfer from its I/O stack location in the IRP.
2. Calls **MmGetMdlVirtualAddress** followed by **MapTransfer**. See Setting Up the System DMA Controller for Packet-Based DMA.
3. Sets up the subordinate device to start the transfer operation.
4. Returns the value **KeepObject**.

Every *AdapterControl* routine must return a system-defined value of type **IO_ALLOCATION_ACTION**. For drivers that use system DMA, the *AdapterControl* routine must return the value **KeepObject**. This allows the driver to retain "ownership" of the system DMA controller and allocated map registers until it has transferred all the requested data.

Because an *AdapterControl* routine cannot wait for the subordinate device to carry out the DMA operation, each *AdapterControl* routine must, at a minimum, do the following:

1. Save context information, particularly the *MapRegisterBase* handle, in the driver's device extension, controller extension, or other driver-accessible resident storage area (nonpaged pool allocated by the driver).

2. Return **KeepObject**.

For additional information, see Writing AdapterControl Routines.

Another driver routine (probably the *DpcForIsr* routine) must call **FlushAdapterBuffers** when each DMA transfer operation is complete. This routine also must call **MapTransfer** and **FlushAdapterBuffers** again if it is necessary to set up the DMA controller more than once to satisfy the current IRP's transfer request.

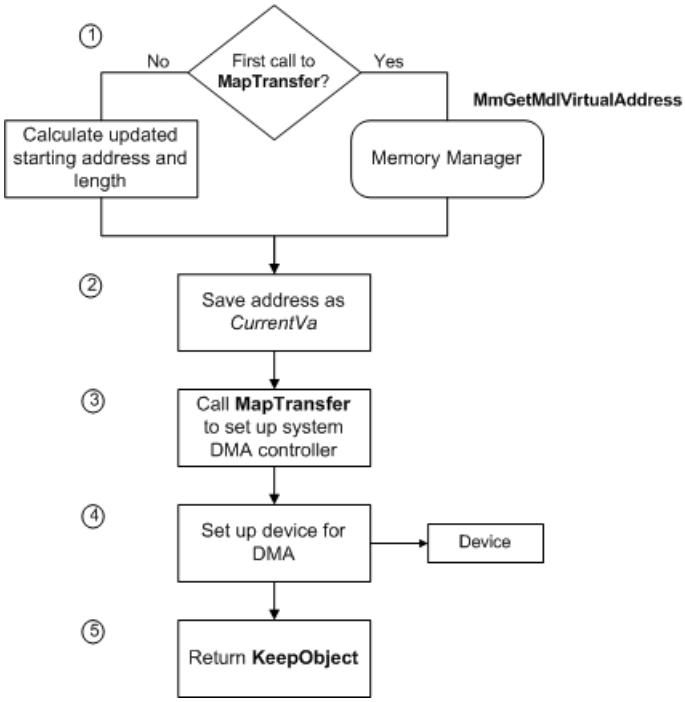
When a driver has satisfied the current IRP's request, it must call **FreeAdapterChannel**. This support routine should be called immediately following the last call to **FlushAdapterBuffers** for the current IRP so that the system DMA controller can be made available for use (by any driver) to satisfy other transfer requests expeditiously.

The driver of a subordinate device with scatter/gather capabilities should also return **KeepObject** from its *AdapterControl* routine. The device must be capable of waiting while the system DMA controller is reprogrammed between DMA operations when the driver must split up a given DMA request. On some Windows platforms, such devices can transfer at most a page of data per DMA operation because the HAL can assign only a single map register to the driver of that device.

Setting Up the System DMA Controller for Packet-Based DMA

6/25/2019 • 3 minutes to read • Edit Online

When **AllocateAdapterChannel** transfers control to a driver's *AdapterControl* routine, the driver "owns" the system DMA controller and a set of map registers. Then, the driver must set up the DMA controller for a transfer operation, as shown in the following figure.



If the driver has a *StartIo* routine, **AllocateAdapterChannel** passes a pointer to **DeviceObject->CurrentIrp** in the *PIrp* parameter to the *AdapterControl* routine. If, however, the driver manages its own queue of IRPs, the driver should include a pointer to the current IRP as part of the context it passes to *AdapterControl*.

As the previous figure shows, the driver's *AdapterControl* routine sets up the DMA transfer, as follows:

1. The *AdapterControl* routine gets the address at which to start the transfer. For the initial transfer required to satisfy an IRP, the *AdapterControl* routine calls **MmGetMdlVirtualAddress**, passing a pointer to the MDL at **Irp->MdlAddress**, which describes the buffer for this DMA transfer.

   MmGetMdlVirtualAddress returns a virtual address that the driver can use as an index for the system physical address where the transfer should start.

   If the IRP requires more than one transfer operation, the driver calculates an updated starting address, as described later in this section.

2. The *AdapterControl* routine saves the address returned by **MmGetMdlVirtualAddress** or calculated in step 1. This address is a required parameter (*CurrentVa*) to **MapTransfer**.

3. The *AdapterControl* routine calls **MapTransfer** to set up the system DMA controller, supplying the following parameters:

   - The adapter object pointer returned by **IoGetDmaAdapter**
   - A pointer (*Mdl*) to the MDL at **Irp->MdlAddress** for the current IRP

- The *MapRegisterBase* handle passed to the driver's *AdapterControl* routine by **AllocateAdapterChannel**
- The value (*CurrentVa*) returned by **MmGetMdlVirtualAddress** if this is the first call to **MapTransfer** for the IRP

  Otherwise, the driver supplies an updated *CurrentVa* value, indicating where in the buffer the next transfer operation should start.
- A pointer to a variable (*Length*) indicating the number of bytes for this transfer

  If the driver can transfer all the requested data with a single call to **MapTransfer** and has no device-specific constraints on its DMA operations, *Length* can be set to the value of **Length** in the driver's I/O stack location of the IRP. At most, the length in bytes can be (PAGE_SIZE * the *NumberOfMapRegisters* returned by **IoGetDmaAdapter**). Otherwise, the driver must split up the request, as explained in Splitting Transfer Requests, and must update the value of *Length* in subsequent calls to **MapTransfer** for the current IRP.
- A Boolean value (*WriteToDevice*), indicating the direction of the transfer operation (TRUE to transfer data from system memory to the device)

MapTransfer returns a logical address. Drivers that use system DMA must ignore this value.

4. The *AdapterControl* routine sets up the device for the DMA operation.
5. The *AdapterControl* routine returns **KeepObject**.

When the device indicates that its current DMA operation has completed, the driver should call **FlushAdapterBuffers**, usually from the driver's *DpcForIsr* routine.

The *DpcForIsr* routine or another driver routine that completes a DMA operation calls **FlushAdapterBuffers** to ensure that any data cached in the system DMA controller is read into system memory or written out to the device. The same routine also must call **MapTransfer** again if it is necessary to reprogram the system DMA controller to transfer more data for the current IRP. Similarly, it must call **FlushAdapterBuffers** again following each transfer operation.

If a driver must call **MapTransfer** more than once for the current IRP, it supplies the same adapter object pointer, *Mdl* pointer, *MapRegisterBase* handle, and transfer direction in every call. However, the driver must update the *CurrentVa* and *Length* parameters before it makes the second and any subsequent calls to **MapTransfer**. To calculate an updated value for each of these parameters, use the following formulas:

- *CurrentVa* = *CurrentVa* + (*Length* requested in the preceding call to **MapTransfer**)
- *Length* = Minimum (remaining **Length** to be transferred, (PAGE_SIZE * *NumberOfMapRegisters* returned by **IoGetDmaAdapter**))

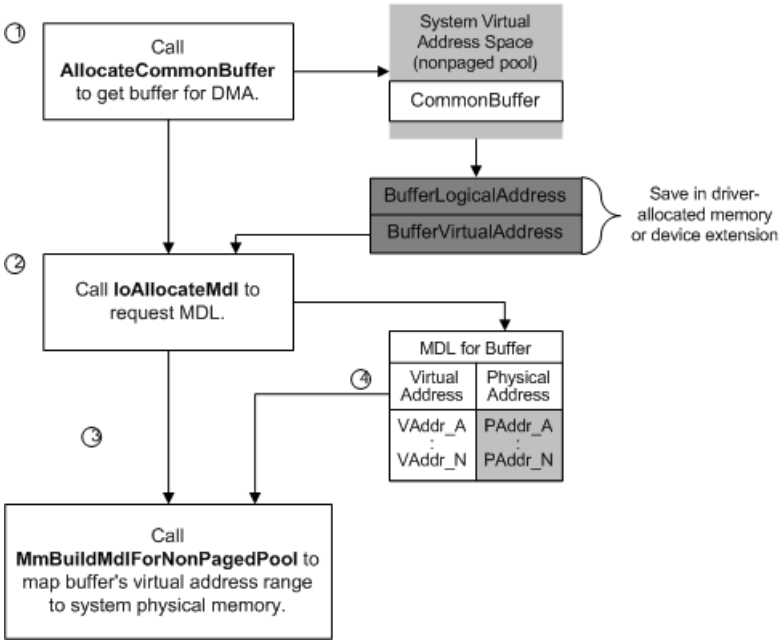
The context information each driver should maintain about its DMA transfers depends on the needs of its particular device. Typical context might include the current virtual address in the MDL (*CurrentVa*), the number of bytes transferred so far, the number of bytes remaining to transfer, possibly a pointer to the current IRP, and any other information the driver writer deems useful.

When the requested transfer is complete, or if the driver must return an error status for the IRP, the driver should call **FreeAdapterChannel** promptly to release the system DMA controller for other drivers and this driver to use.

Using Common-Buffer System DMA

6/25/2019 • 2 minutes to read • Edit Online

A driver that uses a system DMA controller's auto-initialize mode must allocate memory for a buffer into which or from which DMA transfers can be carried out. The driver calls **AllocateCommonBuffer** to get this buffer, typically from the *DispatchPnP* routine that handles an **IRP_MN_START_DEVICE** request. The following figure shows how a driver allocates the buffer and maps its virtual address range to system physical memory.



As the previous figure shows, a driver takes the following steps to allocate a buffer for system DMA:

1. The driver calls **AllocateCommonBuffer**, passing a pointer to the adapter object that was returned by **IoGetDmaAdapter**, along with the length in bytes requested for its buffer. To use memory economically, the input *Length* value for the buffer should either be less than or equal to PAGE_SIZE or should be an integral multiple of PAGE_SIZE.

2. If **AllocateCommonBuffer** returns a **NULL** pointer, the driver should free any system resources it has already claimed and return STATUS_INSUFFICIENT_RESOURCES in response to the **IRP_MN_START_DEVICE** request.

   Otherwise, **AllocateCommonBuffer** allocates the requested amount of memory in system virtual address space and returns two different types of pointers to that buffer:

   - The *LogicalAddress* of the buffer (BufferLogicalAddress in the previous figure), for which the driver must provide storage but which it should ignore thereafter
   - The virtual address of the buffer (BufferVirtualAddress in the previous figure), which the driver also must store so that it can build an MDL describing its buffer for DMA operations

   The driver should store these pointers in the device extension or other driver-allocated resident memory.

3. The driver calls **IoAllocateMdl** to allocate an MDL for the buffer. The driver passes the *VirtualAddress* of the buffer returned by **AllocateCommonBuffer** and the *Length* of its buffer to allocate an MDL.

4. The driver calls **MmBuildMdlForNonPagedPool** with the pointer returned by **IoAllocateMdl** to map the virtual address range for its resident buffer to system physical memory.

After allocating a common buffer and mapping its virtual address range, the driver of a subordinate device can begin to process an IRP that requests a DMA transfer. To do so, the driver calls the following general sequence of support routines:

1. At the driver writer's discretion, **RtlMoveMemory** to copy data from a locked-down user buffer into the driver-allocated common buffer for a transfer to the device
2. **AllocateAdapterChannel** when the driver is ready to program its device for DMA and needs the system DMA controller
3. **MapTransfer**, with the MDL that describes the driver-allocated common buffer, to set up the system DMA controller for the transfer operation

   Note that the driver calls **MapTransfer** only once to set up the system DMA controller to use its common buffer. During a transfer, the driver can call **ReadDmaCounter** to determine how many bytes remain to be transferred, and if necessary, call **RtlMoveMemory** to copy more data to or from a user buffer.
4. **FlushAdapterBuffers** when the driver has completed its DMA transfer to/from the subordinate device
5. **FreeAdapterChannel** as soon as all the requested data has been transferred or if the driver must fail the IRP because of a device I/O error

The adapter object pointer returned by **IoGetDmaAdapter** is a required parameter to each of these support routines except **RtlMoveMemory**.

Individual drivers call this sequence of support routines at different points, depending on how each driver is implemented to service its device. For example, one driver's *StartIo* routine might make the call to **AllocateAdapterChannel**, another driver might make this call from a routine that removes IRPs from a driver-created interlocked queue, and still another driver might make this call when its subordinate DMA device indicates it is ready to transfer data.

Allocating an Adapter Channel for Common-Buffer System DMA

6/25/2019 • 2 minutes to read • Edit Online

A driver calls **AllocateAdapterChannel** after its *DispatchRead* or *DispatchWrite* routine (or any other dispatch routine that handles a DMA transfer) has checked the validity of the IRP's parameters, possibly queued one or more IRPs to another driver routine for further processing, and possibly loaded its common buffer with data to be transferred, if appropriate.

The driver routine that calls **AllocateAdapterChannel** must be executing at IRQL = DISPATCH_LEVEL. The **AllocateAdapterChannel** routine queues the driver's *AdapterControl* routine, which runs after the system DMA controller has been assigned to this driver and a set of map registers has been allocated for the driver's DMA operation.

On entry, the *AdapterControl* routine is given pointers to the device object and context passed in the call to **AllocateAdapterChannel**, as well as a handle for the allocated map registers. The *AdapterControl* routine also is given a pointer to the **DeviceObject->CurrentIrp** if the driver has a *StartIo* routine. If the driver manages its own queuing of IRPs instead of having a *StartIo* routine, the driver should include a pointer to the current IRP as part of the context data it passes when it calls **AllocateAdapterChannel**.

The *AdapterControl* routine typically does the following:

1. Saves or initializes whatever context the driver maintains about DMA operations. The context might include the input *MapRegisterBase* handle the driver must pass to **MapTransfer** and **FlushAdapterBuffers** and, possibly, the **Length** of the requested transfer from its I/O stack location in the IRP.
2. Sets up the subordinate device to start the transfer operation.
3. Returns the value **KeepObject**.

For additional information, see Writing AdapterControl Routines.

For drivers that use a system DMA controller's auto-initialize mode, the *AdapterControl* routine must return the value **KeepObject**. This allows the driver to retain "ownership" of the system DMA controller and allocated map register(s) until it has transferred all the data.

Because an *AdapterControl* routine cannot wait for the subordinate device to carry out the DMA operation, the *AdapterControl* routine must at least do the following:

1. Save context information, particularly the *MapRegisterBase* handle, in the driver's device extension, controller extension, or other driver-accessible resident storage area (nonpaged pool allocated by the driver).
2. Return **KeepObject**.

Another driver routine (probably the *DpcForIsr* routine) must call **FlushAdapterBuffers** and **FreeAdapterChannel** when the DMA transfer operation is complete.

Setting Up the System DMA Controller for Common-Buffer DMA

6/25/2019 • 2 minutes to read • Edit Online

When **AllocateAdapterChannel** transfers control to a driver's *AdapterControl* routine, the driver "owns" the system DMA controller and a set of map registers. Then, the driver must call **MapTransfer** to set up the system DMA controller to use the driver-allocated common buffer before the driver sets up its device for the transfer operation.

The driver supplies the following parameters to **MapTransfer**:

- The adapter object pointer returned by **IoGetDmaAdapter**
- A pointer to the MDL describing the driver-allocated common buffer
- The *MapRegisterBase* handle passed to the driver's *AdapterControl* routine by **AllocateAdapterChannel**
- A pointer to a variable (*Length*) indicating the size in bytes of the driver-allocated common buffer
- A Boolean value, indicating the direction of the transfer operation (TRUE for a requested transfer from system memory to the device)

MapTransfer returns a logical address, which drivers that use system DMA must ignore. When **MapTransfer** returns control, the driver should set up its device for the DMA operation. The driver calls **MapTransfer** only once but continues to copy data between its common buffer and a locked-down user buffer until the requested transfer is done.

The driver can call **ReadDmaCounter** to determine how many bytes currently remain to be transferred in the common buffer; the driver can then continue to fill its common buffer with user data or copy data from its common buffer to the user buffer, depending on the direction of the DMA operation.

When the transfer is complete or if the driver must return an error status for the IRP, the driver calls **FlushAdapterBuffers** to ensure that any data cached in the system DMA controller is read into system memory or written out to the device. Then the driver should call **FreeAdapterChannel** promptly to release the system DMA controller for use by any driver (including itself).

Using Bus-Master DMA

12/5/2018 • 2 minutes to read • Edit Online

Drivers of bus-master DMA devices can use the following kinds of system-supplied DMA support:

- Packet-based DMA if the bus-master adapter allows the driver to determine when a DMA transfer operation is done and/or when to begin another transfer operation for a given IRP. See Using Packet-Based Bus-Master DMA for details.
- Common-buffer DMA (also called *continuous DMA*) if the bus-master adapter does not provide a way for the driver to determine readily when a transfer operation will begin or when a transfer is complete, or if a single buffer area is used continuously or repeatedly for DMA transfers. See Using Common-Buffer Bus-Master DMA for details.

Depending on the nature of the bus-master adapter, some drivers use packet-based DMA exclusively, some use common-buffer DMA exclusively, and some use both. For example, the driver of a bus-master adapter that uses a mailbox scheme to communicate status information and commands might use a common buffer for the mailboxes shared between the driver and its adapter, together with packet-based DMA for data transfers.

Using Packet-Based Bus-Master DMA

6/25/2019 • 2 minutes to read • Edit Online

To use packet-based DMA, drivers of bus-master DMA devices call the following general sequence of support routines as they process an IRP requesting a DMA transfer:

- **KeFlushIoBuffers** just before attempting to allocate map registers for a transfer request (for more information, see Maintaining Cache Coherency)
- **AllocateAdapterChannel** when the driver is ready to program the bus-master adapter for DMA
- **MmGetMdlVirtualAddress** to get an index into the MDL, required as an initial parameter to **MapTransfer**, and **MapTransfer** to make the system physical memory that backs the IRP's buffer device-accessible

  Note that any driver might need to carry out more than one transfer operation in order to satisfy the current IRP, as explained in Splitting Transfer Requests. Drivers of devices that do not have scatter/gather capabilities can call **MapTransfer** once per transfer operation. Drivers of devices that have scatter/gather capabilities can call **MapTransfer** more than once to set up each transfer operation. Alternatively, these drivers can use the system's built-in scatter/gather support, described in Using Scatter/Gather DMA.
- **FlushAdapterBuffers** at the end of each DMA transfer operation to/from the target device, in order to determine whether all the requested data has been completely transferred
- **FreeMapRegisters** as soon as all DMA operations for the current IRP are done, because all the requested data has been completely transferred or because the driver must fail the IRP due to a device or bus I/O error

The adapter object pointer returned by **IoGetDmaAdapter** is a required parameter to **AllocateAdapterChannel**, **MapTransfer**, **FlushAdapterBuffers**, and **FreeMapRegisters**. Note that in versions of Windows NT prior to Windows 2000, bus-master devices could pass a **NULL** adapter object pointer to **MapTransfer** and **FlushAdapterBuffers**. In Windows 2000 and later, drivers can no longer do so.

KeFlushIoBuffers and **MmGetMdlVirtualAddress** require a pointer to the MDL at **Irp->MdlAddress**.

Individual drivers call this sequence of support routines at different points, depending on how each driver is implemented to service its device. For example, one driver's *StartIo* routine might make the call to **AllocateAdapterChannel**, while another driver might make this call from a routine that removes IRPs from a driver-created interlocked queue or device queue.

Instead of using the routines described in this section, any driver that uses packet-based DMA can use support routines intended to streamline scatter/gather DMA, regardless of whether its device has built-in scatter/gather support. See Using Scatter/Gather DMA for details.

Allocating the Bus-Master Adapter Object

6/25/2019 • 3 minutes to read • Edit Online

To prepare for packet-based, bus-master DMA, a driver calls **KeFlushIoBuffers** and **AllocateAdapterChannel** after receiving an **IRP_MJ_READ** or **IRP_MJ_WRITE**. Before the driver calls these routines, its dispatch routine should check the validity of the IRP's parameters. It might also queue the IRP to another driver routine for further processing. The transfer request is the current IRP requiring a device I/O operation.

The driver routine that calls **AllocateAdapterChannel** must be executing at IRQL = DISPATCH_LEVEL. Along with a pointer to the adapter object returned by **IoGetDmaAdapter**, a driver must supply the following when it calls **AllocateAdapterChannel**:

- A pointer to the target device object for the current IRP
- The entry point for its *AdapterControl* routine
- A pointer to any driver-determined context information the *AdapterControl* routine will use

AllocateAdapterChannel queues the driver's *AdapterControl* routine, which runs when the adapter object is free and a set of map registers has been allocated for the driver's DMA operations to or from the target device.

On entry, an *AdapterControl* routine is given the *DeviceObject* and *Context* pointers passed in the call to **AllocateAdapterChannel**, as well as a handle (*MapRegisterBase*) for the allocated map registers.

The *AdapterControl* routine also is given a pointer to the **DeviceObject->CurrentIrp** if the driver has a *StartIo* routine. If the driver manages its own queuing of IRPs instead of having a *StartIo* routine, the driver should include a pointer to the current IRP as part of the context it passes when it calls **AllocateAdapterChannel**.

For the driver of a bus-master DMA device without scatter/gather capabilities, the *AdapterControl* routine usually does the following:

1. Saves or initializes whatever context the driver maintains about DMA operations. The context might include the input *MapRegisterBase* handle the driver must pass to **MapTransfer** and **FlushAdapterBuffers**, the **Length** in bytes of the requested transfer from its I/O stack location in the IRP, and so forth.
2. Calls **MmGetMdlVirtualAddress** followed by **MapTransfer** (described in Setting Up a Transfer Operation, next) to get the logical address its device can use to start the transfer operation.
3. Sets up the bus-master adapter to start the transfer operation.
4. Returns the value **DeallocateObjectKeepRegisters**.

For the driver of a bus-master device with scatter/gather capabilities, the *AdapterControl* routine usually does the following:

1. Saves or initializes whatever state the driver maintains about DMA operations, such as saving the *MapRegisterBase* handle the driver must pass to **MapTransfer** and **FlushAdapterBuffers**, the **Length** in bytes of the requested transfer from its I/O stack location in the IRP, and so forth.
2. Calls **MmGetMdlVirtualAddress** followed by **MapTransfer** (described in the next subsection) to get the logical address its device can use to start the transfer operation.

   The *AdapterControl* routine calls MapTransfer repeatedly until it has used all the available map registers to build a scatter/gather list for the bus-master adapter.
3. Sets up the bus-master adapter to start the transfer operation.

4. Returns the value **DeallocateObjectKeepRegisters**.

For additional information, see Writing AdapterControl Routines.

Note that drivers that perform bus-master DMA can use the **GetScatterGatherList** and **PutScatterGatherList** routines regardless of whether their devices support scatter/gather DMA. Using these routines changes the requirements for the driver's *AdapterControl* routine; see Using Scatter/Gather DMA for details.

An *AdapterControl* routine must return a system-defined value of type IO_ALLOCATION_ACTION. For drivers that use bus-master DMA, the *AdapterControl* routine should typically return the value **DeallocateObjectKeepRegisters**, which allows the driver to retain the allocated map registers for the target device object until it has transferred all the requested data for the current IRP. After the transfer is complete, the DPC routine should call **FreeMapRegisters** to free the allocated map registers. In cases where the device does not support command queuing, however, the *AdapterControl* routine can return **KeepObject** when the transfer for the current IRP is complete, and the DPC routine can call **FreeAdapterChannel** instead.

An *AdapterControl* routine cannot wait for the bus-master adapter to complete a DMA operation.

Regardless of whether the bus-master adapter supports scatter/gather, the *AdapterControl* routine must at least do the following:

1. Save necessary context information—particularly the *MapRegisterBase* handle—in the driver's device extension, controller extension, or other driver-accessible resident storage area (nonpaged pool, allocated by the driver). The driver must pass this handle when it calls **MapTransfer** and **FlushAdapterBuffers**.
2. Return **DeallocateObjectKeepRegisters**.

Another driver routine (probably the *DpcForIsr* routine) must call **FlushAdapterBuffers** when each DMA transfer operation is done. This routine also must set up any additional DMA operations necessary to satisfy the current IRP.

When the driver has satisfied the current IRP's transfer request or must fail the IRP due to a device or bus I/O error, it must call **FreeMapRegisters**. This call should occur immediately following the last call to **FlushAdapterBuffers** for the current IRP, so that the driver can service other DMA requests, possibly for other devices on the bus.

Setting Up a Transfer Operation

6/25/2019 • 4 minutes to read • Edit Online

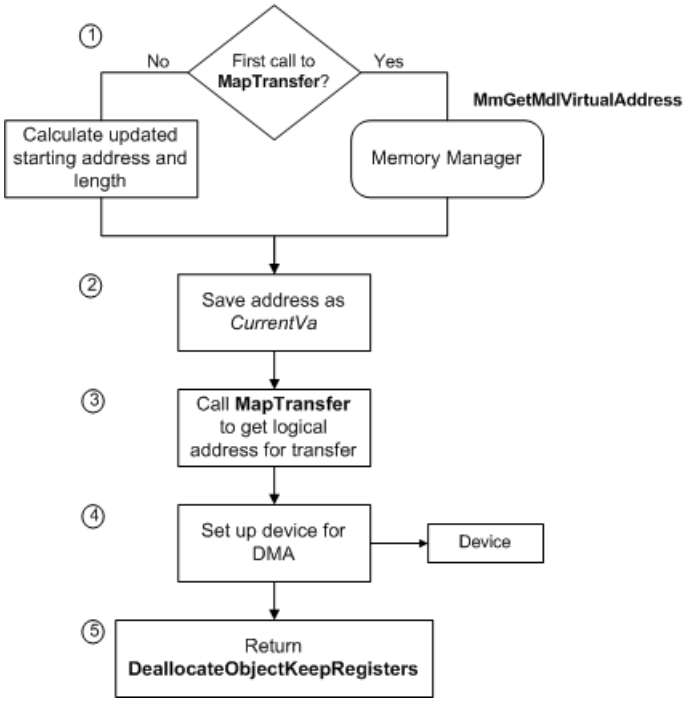
When **AllocateAdapterChannel** transfers control to a driver's *AdapterControl* routine, it has allocated a set of map registers. However, the driver must map system physical memory for the current IRP's transfer request to the bus-master adapter's logical address range, as follows:

1. Call **MmGetMdlVirtualAddress** with the MDL at **Irp->MdlAddress** to get an index for the system physical address where the transfer should start.

   The return value is a required parameter (*CurrentVa*) to **MapTransfer**.

2. Call **MapTransfer** to map the system physical address ranges for the IRP's buffer to the bus-master adapter's logical address range.

The driver can then set up the adapter for the transfer operation. The following figure shows the steps involved in setting up the transfer.



As the previous figure shows, a driver's *AdapterControl* routine sets up a bus-master DMA operation as follows:

1. The *AdapterControl* routine gets the address at which to start the transfer. For the initial transfer required to satisfy an IRP, the *AdapterControl* routine calls **MmGetMdlVirtualAddress**, passing a pointer to the MDL at **Irp->MdlAddress**, which describes the buffer for this DMA transfer.

   MmGetMdlVirtualAddress returns a virtual address that the driver can use as an index for the system physical address where the transfer should start.

   If the IRP requires more than one transfer operation, the driver calculates an updated starting address, as described later in this section.

2. The *AdapterControl* routine saves the address returned by **MmGetMdlVirtualAddress** or calculated in Step 1. This address is a required parameter (*CurrentVa*) to **MapTransfer**.

3. The *AdapterControl* routine calls **MapTransfer**, which returns a logical address at which the driver can program the bus-master adapter to begin the transfer operation. In the call to **MapTransfer**, the driver

supplies the following parameters:

- The adapter object pointer returned by **IoGetDmaAdapter**
- A pointer to the MDL at **Irp->MdlAddress** for the current IRP
- The *MapRegisterBase* handle passed to the driver's *AdapterControl* routine by **AllocateAdapterChannel** (see Allocating the Bus-Master Adapter Object)
- The value returned by **MmGetMdlVirtualAddress** if this is the first call to **MapTransfer** for the current IRP

  Otherwise, the driver supplies an updated *CurrentVa* value, indicating the next physical-to-logical mapping to be done. (How to calculate an updated *CurrentVa* is described later in this section.)
- A pointer to a variable (*Length*), which indicates the number of bytes for this transfer

  If the driver has enough map registers to transfer all the requested data in a single DMA operation and has no device-specific constraints on its DMA operations, the *Length* can be set to the value of **Length** in the driver's I/O stack location of the IRP. At most, the input length in bytes can be (PAGE_SIZE * the *NumberOfMapRegisters* returned by **IoGetDmaAdapter**). Otherwise, the driver must split up the request, as explained in Splitting Transfer Requests, and must update the value of *Length* in subsequent calls to **MapTransfer** for the current IRP.
- A Boolean value (*WriteToDevice*), indicating the direction of the transfer operation (TRUE to transfer data from memory to the device)

4. The *AdapterControl* routine sets up the device for the DMA operation.
5. The *AdapterControl* routine returns **DeallocateObjectKeepRegisters**.

If the driver must call **MapTransfer** more than once to satisfy the current IRP, it supplies the same adapter object pointer, *Mdl* pointer, *MapRegisterBase* handle, and transfer direction in every call to **MapTransfer**. However, the driver must supply updated *CurrentVa* and *Length* values in its second and subsequent calls to **MapTransfer**. Use the following formulas to calculate these values:

- *CurrentVa* = *CurrentVa* + (*Length* requested in preceding call to **MapTransfer**)
- *Length* = Minimum (remaining **Length** to be transferred, (PAGE_SIZE * *NumberOfMapRegisters* returned by **IoGetDmaAdapter**))

The context information each driver should maintain about its DMA transfers depends on the needs of its particular device. Typical context might include the current virtual address in the MDL (*CurrentVa*), the number of bytes transferred so far, the number of bytes remaining to transfer, and possibly a pointer to the current IRP.

For drivers of devices with scatter/gather capabilities, the *Length* parameter to **MapTransfer** is both an input and output parameter. On return from **MapTransfer**, it indicates how many bytes of data the system has mapped. That is, the return value of *Length*, in combination with the returned logical address, indicates the range of logical addresses the bus-master adapter can use for this piece of the transfer in this DMA operation.

Note Since *Length* is overwritten by **MapTransfer**, follow this implementation guideline: Never pass a pointer to the **Length** in the driver's I/O stack location of an IRP as the *Length* parameter to **MapTransfer** if your device supports scatter/gather.

Doing this could destroy the value in the current IRP, making it impossible to determine whether the driver has transferred all the requested data.

At the end of each DMA operation, the driver must call **FlushAdapterBuffers** with a valid adapter object pointer and the *MapRegisterBase* handle to be sure that all the data has been transferred, and to release the physical-to-logical mappings for the current DMA operation. If the driver must set up additional DMA operations to satisfy the

current IRP, it must call **FlushAdapterBuffers** after each transfer operation is complete.

When all the requested transfer is complete or the driver must return an error status for the IRP, the driver should call **FreeMapRegisters** immediately after its last call to **FlushAdapterBuffers** in order to get the best possible throughput for the bus-master adapter. In its call to **FreeMapRegisters**, the driver must pass the adapter object pointer that it passed in the preceding call to **AllocateAdapterChannel**.

Using Common-Buffer Bus-Master DMA

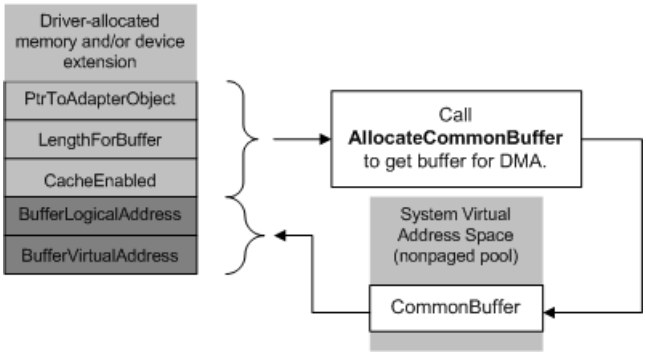
6/25/2019 • 2 minutes to read • Edit Online

As described in Using Bus-Master DMA, some drivers for bus-master DMA devices use common-buffer DMA exclusively, and some use common-buffer DMA in combination with packet-based DMA.

Use common-buffer DMA economically. Setting up a common buffer can tie up some (or all, depending on the size of the requested buffer) of the map registers associated with the adapter object that represents the bus-master adapter.

Setting up common-buffer areas economically, such as by using **PAGE_SIZE** chunks or a single allocation, leaves more map registers available for packet-based DMA operations. It also leaves more system memory free for other purposes, which produces better overall driver and system performance.

To set up a common buffer for bus-master DMA, a bus-master DMA device driver must call **AllocateCommonBuffer** with the adapter object pointer returned by **IoGetDmaAdapter**. Typically, a driver makes this call from its *DispatchPnP* routine for **IRP_MN_START_DEVICE** requests. A driver should allocate a common buffer only if it will use the buffer repeatedly for its DMA operations while the driver remains loaded. The following diagram illustrates such a call to **AllocateCommonBuffer**.



The requested size for the buffer, shown in the previous diagram as LengthForBuffer, determines how many map registers must be used to provide a virtual-to-logical mapping for the common buffer. Use the **BYTES_TO_PAGES** macro to determine the maximum number of pages needed (**BYTES_TO_PAGES** (*LengthForBuffer*)). This value cannot be greater than the *NumberOfMapRegisters* returned by **IoGetDmaAdapter**.

In addition, the caller must supply the following:

- A Boolean value that indicates whether caching should be enabled

  Note This value is ignored. The operating system determines whether to enable cached memory in the common buffer that is to be allocated. That decision is based on the processor architecture and device bus.

  On computers with x86-based, x64-based, and Itanium-based processors, cached memory is enabled.

  On computers with ARM or ARM 64-based processors, the operating system does not automatically enable cached memory for all devices. The system relies on the ACPI_CCA method for each device to determine whether the device is cache-coherent.

- A pointer to a driver-defined variable that will contain the device-accessible base *Logical Address* for the buffer (BufferLogicalAddress in the previous diagram) on return from **AllocateCommonBuffer**

If the call succeeds, **AllocateCommonBuffer** returns a driver-accessible base virtual address for the buffer (BufferVirtualAddress in the previous diagram), which the driver must save in its device extension, controller

extension, or other driver-accessible resident storage area (nonpaged pool allocated by the driver).

AllocateCommonBuffer returns **NULL** if it cannot allocate memory for the buffer. If the returned base virtual address is **NULL**, the driver either must use the system's packet-based DMA support exclusively or the driver must fail the **IRP_MN_START_DEVICE** request, returning STATUS_INSUFFICIENT_RESOURCES.

Otherwise, the driver can use the allocated common buffer as a driver- and adapter-accessible storage area for DMA transfers.

When the PnP manager sends an IRP that stops or removes the device, the driver must call **FreeCommonBuffer** to release each common buffer it has allocated.

Version 3 of the DMA Operations Interface

6/25/2019 • 2 minutes to read • Edit Online

Version 3 of the DMA operations interface is available starting with Windows 8. The **DMA_OPERATIONS** structure for this interface contains a number of new routines that are not defined in previous versions of this interface. For a list of the routines in version 3, see **DMA_OPERATIONS**.

Although version 3 of the DMA operations interface is available across all Windows hardware platforms, this interface has many features to enable kernel-mode drivers to use the advanced capabilities of system DMA controllers in System on a Chip (SoC) integrated circuits. These capabilities typically include the ability to do scatter/gather DMA transfers. In contrast, previous versions of the DMA operations interface restrict scatter/gather DMA transfers to bus-master devices. The version-3 interface simplifies the management of scatter/gather lists and reduces the need for driver intervention during complex DMA transfers.

To use version 3 of the DMA operations interface to perform a DMA transfer, a driver typically calls the following routines:

IoGetDmaAdapter
Allocates a DMA adapter object and returns a pointer to a **DMA_ADAPTER** structure that contains the DMA operations interface.

GetDmaTransferInfo
Provides a description of the resources that are required to perform the DMA transfer that is described by the caller.

AllocateAdapterChannelEx
Allocates the resources required for the DMA transfer and assigns these resources to the DMA adapter object.

MapTransferEx
Initializes the map registers and the scatter/gather buffer for the DMA transfer, and starts the transfer.

FlushAdapterBuffersEx
Performs any cache operations that might be required at the end of the DMA transfer.

FreeAdapterChannel
Frees the DMA channel and map registers.

PutDmaAdapter
Releases the adapter object.

These routines are used both for bus-master devices that use dedicated DMA controllers, and for subordinate devices that share a system DMA controller. For a step-by-step description of the calls that a driver makes during a typical DMA transfer, see Basic Calling Pattern for Version-3 DMA Routines.

Note In version 3 of the DMA operations interface, calls to the **KeFlushIoBuffers** routine are not required either before or after DMA transfers. The reason is that the following routines cover the need for flushing data caches on platforms that do not enforce cache coherency in hardware:

- **MapTransferEx** ensures that processor data caches are flushed before write (memory-to-device) transfers.
- **FlushAdapterBuffersEx** ensures that caches are invalidated after read (device-to-memory) transfers.

On an x86 or x64 processor, the **KeFlushIoBuffers** call performs no operations, and this call, while unnecessary, does not interfere with the operation of the hardware platform. On an ARM processor, calls to **KeFlushIoBuffers** during DMA transfers perform cache operations that are unnecessary and can degrade performance.

Basic Calling Pattern for Version-3 DMA Routines

6/25/2019 • 6 minutes to read • Edit Online

To perform a DMA transfer that uses the routines in version 3 of the DMA operations interface, your driver should follow the steps described in the following list. These steps are common to both subordinate devices and bus-master devices. Version 3 of this interface is available starting with Windows 8. For more information about the routines in this interface, see **DMA_OPERATIONS**.

Step 1: Obtain a DMA adapter object

In preparation for a DMA transfer, the driver calls the **IoGetDmaAdapter** routine to obtain a DMA adapter object. A DMA adapter object is a software object that represents either a bus-master device, or a request line on a system DMA controller. This object contains the DMA operations interface for the bus that is used to transfer data to or from the device. Additionally, this object synchronizes the driver's access to the shared resources that are required to perform the transfer. For more information, see Introduction to Adapter Objects.

Step 2: Obtain a description of the required DMA resources

The driver calls the **GetDmaTransferInfo** routine to get a description of the DMA resources that it needs to perform the transfer.

The input parameters to this call describe the memory buffer to use for the transfer, and the direction (read or write) of the transfer.

The resource requirements obtained from this call include the number of map registers and the size of the scatter/gather list that is needed to describe the data buffer for the transfer. In the subsequent call to the **AllocateAdapterChannelEx** routine (see step 3), the driver supplies the map register count as an input parameter.

Step 3: Request the required DMA resources

The driver calls the **AllocateAdapterChannelEx** routine to allocate resources to assign to the DMA adapter object. These resources include a DMA channel and map registers.

An **AllocateAdapterChannelEx** call can be asynchronous or synchronous.

If the DMA_SYNCHRONOUS_CALLBACK flag is not set, the call is asynchronous. In this case, the *ExecutionRoutine* parameter points to a caller-supplied execution routine that is called when the requested resources are available. If successful, an asynchronous **AllocateAdapterChannelEx** call returns STATUS_SUCCESS without waiting for the execution routine to run.

If the DMA_SYNCHRONOUS_CALLBACK flag is set, the **AllocateAdapterChannelEx** call is synchronous. In this case, the *ExecutionRoutine* parameter in the call is optional, and **AllocateAdapterChannelEx** behaves as follows:

- If *ExecutionRoutine* is non-NULL, and the DMA resources can be allocated immediately, **AllocateAdapterChannelEx** calls the execution routine in the context of the calling thread. After the execution routine finishes running, **AllocateAdapterChannelEx** returns STATUS_SUCCESS. If the resources are not immediately available, **AllocateAdapterChannelEx** fails and returns error status code STATUS_INSUFFICIENT_RESOURCES.
- If *ExecutionRoutine* is NULL, and **AllocateAdapterChannelEx** can immediately allocate the DMA

resources, **AllocateAdapterChannelEx** returns STATUS_SUCCESS. If all resources are not immediately available, the call fails with error status code STATUS_INSUFFICIENT_RESOURCES.

For synchronous calls that return STATUS_SUCCESS, if the *MapRegisterBase* parameter to **AllocateAdapterChannelEx** is non-NULL, **AllocateAdapterChannelEx** writes the base address of the allocated map registers to the address pointed to by the *MapRegisterBase* parameter. If *ExecutionRoutine* is NULL, *MapRegisterBase* must be non-NULL. If *ExecutionRoutine* is non-NULL, the *MapRegisterBase* parameter to **AllocateAdapterChannelEx** is optional, and the execution routine receives the map register base address as an input parameter.

For asynchronous **AllocateAdapterChannelEx** calls, *ExecutionRoutine* must be non-NULL, and the execution routine receives the map register base address as an input parameter.

In subsequent calls to the **MapTransferEx** routine (see step 5), the driver supplies the map register base address as an input parameter.

If *ExecutionRoutine* is non-NULL, the execution routine returns a status value to indicate the disposition of the allocated resources. For system DMA transfers, this return value must be **KeepObject**. This value informs the operating system that the adapter object (and all of its allocated resources) is in use and should not be freed. If no execution routine is supplied, the driver must instead call the **FreeAdapterObject** routine and supply **KeepObject** as the *AllocationOption* parameter.

Step 4: If necessary, cancel the pending resource request

After an **AllocateAdapterChannelEx** call queues a DMA adapter to wait for DMA resources, the driver can, if necessary, call the **CancelAdapterChannel** routine to cancel the pending resource request.

If **CancelAdapterChannel** returns TRUE, the resource request is successfully canceled. If an execution routine was supplied in the **AllocateAdapterChannelEx** call, this routine does not run.

If **CancelAdapterChannel** returns FALSE, the resource request cannot be canceled because it was already granted. If an execution routine was supplied in the **AllocateAdapterChannelEx** call, this routine will be called.

Step 5: Initialize the DMA resources and start the DMA transfer

The driver calls **MapTransferEx** to initialize the DMA resources and to start the DMA transfer. This call might occur in the same driver thread that calls **AllocateAdapterChannelEx**, or it might occur in the execution routine that the driver supplies to **AllocateAdapterChannelEx**. If more than one **MapTransferEx** call is required to transfer the entire DMA data buffer, a later **MapTransferEx** call might occur in the completion routine for the previous **MapTransferEx** call.

MapTransferEx supports chained MDLs as input parameters. Each MDL describes a region of the DMA buffer that is contiguous in virtual memory. When **MapTransferEx** builds the scatter/gather list, it automatically handles transitions from one virtually contiguous buffer region to the next without driver intervention. For more information, see Using the MapTransferEx Routine.

For a system DMA transfer, a pointer to a DMA completion routine can be passed to **MapTransferEx** in the optional *DmaCompletionRoutine* parameter. This routine is scheduled to run at dispatch level in response to an interrupt from the system DMA controller that indicates that the DMA transfer is complete.

If **MapTransferEx** is unable to map the entire requested transfer size, it will set the **Length* output parameter to the length that was mapped, and return STATUS_SUCCESS.

Step 6: If necessary, perform hardware-specific operations

MapTransferEx returns STATUS_SUCCESS to indicate that the DMA transfer is successfully initiated. On some

platforms, the driver might have to take some additional action, outside of the **MapTransferEx** call, to start the transfer, but this type of delayed start is not required for all platforms. Drivers must not depend on such delays for decisions about using and freeing allocated resources.

The routines in the DMA operations interface maintain cache coherency for DMA transfers in a way that is transparent to the drivers that use these routines. On platforms that do not enforce cache coherency in hardware, **MapTransferEx** ensures that processor data caches are flushed before write (memory-to-device) transfers. For read (device-to-memory) transfers, the caches are invalidated during the call to the **FlushAdapterBuffersEx** routine (see step 8) that follows every **MapTransferEx** call.

Step 7: Receive notification when the DMA transfer finishes

When a DMA transfer completes, the driver is notified in one of these two ways:

- An interrupt to the device driver, for a bus-master device
- Execution of the driver-supplied completion routine, for a subordinate device that uses a system DMA controller

For a system DMA transfer, a driver can supply a completion routine to **MapTransferEx** as an input parameter.

Step 8: Flush any data that remains in the cache

After the DMA transfer completes, the driver must call the **FlushAdapterBuffersEx** routine to flush any data that remains in the cache. The driver must call **FlushAdapterBuffersEx** after every **MapTransferEx** call.

If a **MapTransferEx** call maps only a part of the DMA data buffer, the driver must call **MapTransferEx** again to map the remaining data. A complex transfer might require several **MapTransferEx** calls. For each additional **MapTransferEx** call, repeat steps 5 through 8.

Step 9: Free the DMA channel and map registers

After the entire DMA data buffer is successfully mapped and the final transfer completes, the driver must call the **FreeAdapterChannel** routine to free the DMA channel and any previously allocated map registers.

Step 10: Release the DMA adapter object

After all DMA transfers are complete and any previously allocated map registers are freed, the driver calls the **PutDmaAdapter** routine to release the adapter object.

Using the MapTransferEx Routine

6/25/2019 • 4 minutes to read • Edit Online

The **MapTransferEx** routine initializes a set of previously allocated DMA resources and starts a DMA transfer. This routine is available in version 3 of the DMA operations interface. Version 3 of this interface is supported starting with Windows 8. For more information about the DMA operations interface, see **DMA_OPERATIONS**.

Comparison of MapTransferEx to MapTransfer

MapTransferEx is an improved version of the **MapTransfer** routine. **MapTransfer** is available in all versions of the DMA operations interface, starting with version 1 in Windows 2000. One call to **MapTransfer** can map one contiguous block of physical memory from an MDL. However, the data buffer for a complex DMA transfer might be described by an MDL chain, and each MDL in the chain might describe several blocks of physically contiguous memory. To use **MapTransfer** to transfer such a buffer, a driver must make many calls to **MapTransfer**. Typically, these calls are made inside a pair of nested loops. The inner loop iterates from one block of contiguous physical memory to the next in each MDL, and the outer loop iterates from one MDL to the next in the MDL chain.

In contrast, one call to **MapTransferEx** can transfer the entire data buffer for a complex DMA transfer. The following three **MapTransferEx** parameters describe the buffer memory to use for the transfer.

PARAMETER	DESCRIPTION
Mdl	A pointer to the first MDL in a chain of one or more MDLs. For more information about MDL chains, see Using MDLs.
Offset	The byte offset of the buffer from start of the memory that is described by the MDL chain.
Length	A pointer to a location that contains the length, in bytes, of the data buffer.

At the start of a **MapTransferEx** call, the **MapTransferEx** routine advances through the MDL chain to find the start of the buffer. The start of the buffer is specified by the *Offset* parameter. Next, working from the start of the buffer to the end, **MapTransferEx** constructs a scatter/gather list in which each buffer fragment in the list is a physically contiguous block of memory from the MDL chain. To construct this list, **MapTransferEx** steps from one physically contiguous block of memory to the next within each MDL, and from one MDL to the next in the MDL chain. List construction finishes when the total amount of buffer memory described by the scatter/gather list equals the number of bytes specified by the **Length* input parameter. The ordering of the buffer fragments in the resulting scatter/gather list matches the ordering of the physically contiguous blocks in the MDL chain.

Multiple calls to MapTransferEx

MapTransferEx might not always be able to transfer an entire DMA data buffer in one call. The following list describes some of the conditions that might require **MapTransferEx** to be called more than once to complete the transfer:

- The DMA adapter requires map registers, and the number of map registers assigned to the adapter is not sufficient to describe the entire buffer.

- The storage allocated by the driver to contain the scatter/gather list is not large enough to contain the scatter/gather list for the entire buffer.
- The transfer uses a system DMA controller that limits the number of buffer fragments that can be specified in a hardware scatter/gather list.

In all of these cases, **MapTransferEx** maps as much of the data buffer as it can in one call, and tells the driver how much of the buffer was mapped by the call. The preceding list does not include other conditions, such as platform-specific cache behavior, that might require more than one call to **MapTransferEx** to complete a transfer. Future hardware platforms might impose additional constraints on DMA transfer length. For these reasons, driver developers should design their drivers to correctly handle the case in which **MapTransferEx** cannot map an entire DMA data buffer in one call.

Before calling **MapTransferEx**, the caller sets the **Length* parameter to the number of bytes in the DMA data buffer that still need to be mapped. Before returning, **MapTransferEx** sets **Length* to the number of bytes in the buffer that were actually mapped by the call. When a **MapTransferEx** call cannot map the entire buffer length, as specified by the **Length* input value, the output value of **Length* is less than its input value. If a DMA transfer requires two or more **MapTransferEx** calls, the calling driver must obtain the **Length* output value from one call before it can specify the **Length* input value for the next call.

For example, if a **MapTransferEx** call can transfer only X bytes to or from a buffer for which *Offset* = B and **Length* = N (on input), then, on return, **Length* = X. For the next call to **MapTransferEx**, the driver should set *Offset* = B + X and **Length* = N - X. In both calls, the same MDL chain is used without modification.

If the caller specifies a *DmaCompletionRoutine*, **MapTransferEx** writes the **Length* output value before it schedules the *DmaCompletionRoutine* to run. This behavior ensures that the updated **Length* value is always available before the *DmaCompletionRoutine* runs. For example, if a DMA transfer requires two **MapTransferEx** calls, the *DmaCompletionRoutine* that the first call schedules can obtain the **Length* output value from the first call. The routine can then use this value to calculate the **Length* input value for the second call. Typically, the *Length* parameter points to a location in the **CompletionContext* value that is supplied to the *DmaCompletionRoutine* as a parameter.

PIO Techniques

12/5/2018 • 2 minutes to read • Edit Online

On some computer hardware architectures, the transfer of data from the CPU (central processing unit) to devices is done by Programmed Input/Output (PIO). Using PIO requires that the CPU wait for the data to be transferred, which can become very inefficient. This technology has been replaced in most instances by Direct Memory Access (DMA) because DMA can assign the transfer of the data to a hardware controller, letting the CPU perform other tasks.

For information on using caches with PIO, see Flushing Cached Data during PIO Operations.

Flushing Cached Data during PIO Operations

6/25/2019 • 2 minutes to read • Edit Online

On some platforms, the instruction and data caches in the processor exhibit cache coherency anomalies during PIO read operations.

Note To maintain data integrity during their read operations, drivers that use PIO must follow this guideline: Call **KeFlushIoBuffers** at the end of each read operation.

For example, a driver making a PIO transfer from its device to system memory should call **KeFlushIoBuffers** at the end of each device transfer operation. As another example, a driver that reads a sequence of device registers into system memory should call **KeFlushIoBuffers** at the end of the sequence. Otherwise, such a driver might attempt to access data that is still in the processor's data cache, rather than in system memory, on some platforms.

KeFlushIoBuffers does nothing if the processor can be relied on to maintain cache coherency, so calls to this support routine have almost no overhead in such a platform.

Accessing Device Configuration Space

6/25/2019 • 2 minutes to read • Edit Online

Some buses provide a way of accessing a special configuration space for each device attached to the bus. This section explains how a driver can get information from a target device's configuration space, provided the driver is loaded in the same driver stack as the driver for the target device, either as a function driver or a filter driver.

In Microsoft Windows NT 4.0, drivers get information from a target device's configuration space by scanning the bus and calling the **HalGetBusData** and **HalGetBusDataByOffset** routines. In Windows 2000 and later operating systems, the hardware buses are controlled by their respective bus drivers and not by HAL. Therefore, all of the HAL routines that used to help drivers retrieve bus-related information are obsolete in Windows 2000.

The configuration space for a device contains a description of the device and its resource requirements. On Windows 2000 and later operating systems, a driver does not need to query a device to find resources. The driver gets the resources from the Plug and Play (PnP) manager in its **IRP_MN_START_DEVICE** request. Typically, a well-written driver would not require any of this information to function correctly. If, for some reason, the driver requires this information, the code sample in the Obtaining Device Configuration Information at IRQL = PASSIVE_LEVEL section shows how to get the resources. The driver must be part of the target device's driver stack, because it requires the underlying physical device objects (PDO) of the target device to send the appropriate PnP request.

Obtaining Device Configuration Information at IRQL = PASSIVE_LEVEL

6/25/2019 • 2 minutes to read • Edit Online

To access device configuration space at IRQL = PASSIVE_LEVEL, you must use **IRP_MN_READ_CONFIG** and **IRP_MN_WRITE_CONFIG**. You indicate in the IRP stack which configuration space you wish to access and where the I/O buffer is. See the description of the **IO_STACK_LOCATION** structure for further details.

The following code sample demonstrates how to access a device's configuration space.

```
NTSTATUS
ReadWriteConfigSpace(
    IN PDEVICE_OBJECT  DeviceObject,
    IN ULONG  ReadOrWrite,  // 0 for read, 1 for write
    IN PVOID  Buffer,
    IN ULONG  Offset,
    IN ULONG  Length
    )
{
    KEVENT event;
    NTSTATUS status;
    PIRP irp;
    IO_STATUS_BLOCK ioStatusBlock;
    PIO_STACK_LOCATION irpStack;
    PDEVICE_OBJECT targetObject;

    PAGED_CODE();
    KeInitializeEvent(&event, NotificationEvent, FALSE);
    targetObject = IoGetAttachedDeviceReference(DeviceObject);
    irp = IoBuildSynchronousFsdRequest(IRP_MJ_PNP,
                                       targetObject,
                                       NULL,
                                       0,
                                       NULL,
                                       &event,
                                       &ioStatusBlock);
    if (irp == NULL) {
        status = STATUS_INSUFFICIENT_RESOURCES;
        goto End;
    }
    irpStack = IoGetNextIrpStackLocation(irp);
    if (ReadOrWrite == 0) {
        irpStack->MinorFunction = IRP_MN_READ_CONFIG;
    } else {
        irpStack->MinorFunction = IRP_MN_WRITE_CONFIG;
    }
    irpStack->Parameters.ReadWriteConfig.WhichSpace = PCI_WHICHSPACE_CONFIG;
    irpStack->Parameters.ReadWriteConfig.Buffer = Buffer;
    irpStack->Parameters.ReadWriteConfig.Offset = Offset;
    irpStack->Parameters.ReadWriteConfig.Length = Length;

    // Initialize the status to error in case the bus driver does not
    // set it correctly.
    irp->IoStatus.Status = STATUS_NOT_SUPPORTED;
    status = IoCallDriver(targetObject, irp);
    if (status == STATUS_PENDING) {
        KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
        status = ioStatusBlock.Status;
    }
End:
    // Done with reference
    ObDereferenceObject(targetObject);
    return status;
}
```

Obtaining Device Configuration Information at IRQL = DISPATCH_LEVEL

10/16/2019 • 3 minutes to read • Edit Online

The method illustrated in the Obtaining Device Configuration Information at IRQL = PASSIVE_LEVEL section makes use of I/O request packets (IRPs) and therefore is only valid for drivers running at IRQL = PASSIVE_LEVEL. Drivers running at IRQL = DISPATCH_LEVEL must use a bus interface to obtain device configuration space data. To obtain this data, you can use a bus-specific interface or the system-supplied bus-independent bus interface, **BUS_INTERFACE_STANDARD**.

The GUID_BUS_INTERFACE_STANDARD interface (defined in `wdmguid.h`) enables device drivers to make direct calls to parent bus driver routines instead of using I/O request packets (IRP) to communicate with the bus driver. In particular, this interface enables drivers to access routines that the bus driver provides for the following functions:

- Translating bus addresses
- Retrieving a DMA adapter structure in cases where the bus adapter supports DMA
- Reading and setting the bus configuration space for a particular device on the bus

To use this interface, send an IRP_MN_QUERY_INTERFACE IRP to your bus driver with InterfaceType = GUID_BUS_INTERFACE_STANDARD. The bus driver supplies a pointer to a BUS_INTERFACE_STANDARD structure that contains pointers to the individual routines of the interface.

It is preferable to use **BUS_INTERFACE_STANDARD** where possible, because a bus number is not required to retrieve configuration information when using **BUS_INTERFACE_STANDARD**, whereas drivers must often identify the bus number when retrieving bus-specific interfaces. Bus numbers for some buses, such as PCI, can change dynamically. Therefore, drivers should not depend on the bus number to access the PCI ports directly. Doing so might lead to system failure.

Three steps are required when accessing the configuration space of a PCI device at IRQL = DISPATCH_LEVEL:

1. Send an **IRP_MN_QUERY_INTERFACE** request at IRQL = PASSIVE_LEVEL to get the direct-call interface structure (**BUS_INTERFACE_STANDARD**) from the PCI bus driver. Store this in a nonpaged pool memory (typically in a device extension).
2. Call the **BUS_INTERFACE_STANDARD** interface routines, *SetBusData* and *GetBusData*, to access the PCI configuration space at IRQL = DISPATCH_LEVEL.
3. Dereference the interface. The PCI bus driver takes a reference count on the interface before it returns, so the driver that accesses the interface must dereference it, once it is no longer needed.

The following code sample demonstrates how to implement these three steps:

```
NTSTATUS
GetPCIBusInterfaceStandard(
    IN  PDEVICE_OBJECT DeviceObject,
    OUT PBUS_INTERFACE_STANDARD BusInterfaceStandard
    )
/*++
Routine Description:
    This routine gets the bus interface standard information from the PDO.
Arguments:
    DeviceObject - Device object to query for this information.
    BusInterface - Supplies a pointer to the retrieved information.
Return Value:
    NT status.
--*/
{
    KEVENT event;
    NTSTATUS status;
    PIRP irp;
    IO_STATUS_BLOCK ioStatusBlock;
    PIO_STACK_LOCATION irpStack;
    PDEVICE_OBJECT targetObject;

    Bus_KdPrint(("GetPciBusInterfaceStandard entered.\n"));
    KeInitializeEvent(&event, NotificationEvent, FALSE);
    targetObject = IoGetAttachedDeviceReference(DeviceObject);
    irp = IoBuildSynchronousFsdRequest(IRP_MJ_PNP,
                                       targetObject,
                                       NULL,
                                       0,
                                       NULL,
                                       &event,
                                       &ioStatusBlock);
    if (irp == NULL) {
        status = STATUS_INSUFFICIENT_RESOURCES;
        goto End;
    }
    irpStack = IoGetNextIrpStackLocation( irp );
    irpStack->MinorFunction = IRP_MN_QUERY_INTERFACE;
    irpStack->Parameters.QueryInterface.InterfaceType = (LPGUID)&GUID_BUS_INTERFACE_STANDARD;
    irpStack->Parameters.QueryInterface.Size = sizeof(BUS_INTERFACE_STANDARD);
    irpStack->Parameters.QueryInterface.Version = 1;
    irpStack->Parameters.QueryInterface.Interface = (PINTERFACE)BusInterfaceStandard;
    irpStack->Parameters.QueryInterface.InterfaceSpecificData = NULL;

    // Initialize the status to error in case the bus driver does not
    // set it correctly.
    irp->IoStatus.Status = STATUS_NOT_SUPPORTED;
    status = IoCallDriver(targetObject, irp);
    if (status == STATUS_PENDING) {
        KeWaitForSingleObject(&event, Executive, KernelMode, FALSE, NULL);
        status = ioStatusBlock.Status;
    }
End:
    // Done with reference
    ObDereferenceObject(targetObject);
    return status;
}
```

The following code snippet shows how to use the *GetBusData* interface routine to get the configuration space data (step 2).

```
bytes = busInterfaceStandard.GetBusData(
                    busInterfaceStandard.Context,
                    PCI_WHICHSPACE_CONFIG,
                    Buffer
                    Offset,
                    Length);
```

When the driver is done with the interface, it can use code similar to the following snippet to dereference the interface (step 3). Drivers must not call interface routines after dereferencing the interface.

```
    (busInterfaceStandard.InterfaceDereference)(
                    (PVOID)busInterfaceStandard.Context);
```

The interface synchronizes the caller's access to the bus hardware with the PCI bus driver's access. The driver writer need not worry about creating spin locks to avoid contending with the PCI bus driver for access to bus hardware.

Note, that if all that is needed are bus, function, and device numbers, it is usually unnecessary to resort to a bus interface to obtain this information. This data can be retrieved indirectly by passing the PDO of the target device to the **IoGetDeviceProperty** function as follows:

```
    ULONG   propertyAddress, length;
    USHORT  FunctionNumber, DeviceNumber;

    // Get the BusNumber. Be warned that bus numbers may be
    // dynamic and therefore subject to change unpredictably!!!
    IoGetDeviceProperty(PhysicalDeviceObject,
                        DevicePropertyBusNumber,
                        sizeof(ULONG),
                        (PVOID)&BusNumber,
                        &length);

    // Get the DevicePropertyAddress
    IoGetDeviceProperty(PhysicalDeviceObject,
                        DevicePropertyAddress,
                        sizeof(ULONG),
                        (PVOID)&propertyAddress,
                        &length);

    // For PCI, the DevicePropertyAddress has device number
    // in the high word and the function number in the low word.
    FunctionNumber = (USHORT)((propertyAddress) & 0x0000FFFF);
    DeviceNumber = (USHORT)(((propertyAddress) >> 16) & 0x0000FFFF);
```

Obtaining Configuration Information from Other Driver Stacks

7/9/2019 • 2 minutes to read • Edit Online

At times you need to obtain information from the configuration space of a device whose driver is on a stack other than the one that your driver is on. For instance, suppose you want to set a bit in the configuration space of a PCI-to-PCI bridge and you do not have a pointer to the PDO of the bridge. Although the operating system enumerates PCI-to-PCI bridges and creates a PDO for every bridge on the system, it does not register device interfaces for these devices. Therefore, you cannot use the device interface mechanism to access the configuration space of these bridges. For more information about device interfaces see Introduction to Device Interfaces.

In Windows NT 4.0, drivers could access the configuration space of such devices using the **HalGetBusData** and **HalSetBusData** routines. In Windows 2000 and later versions of Windows, this is no longer the case.

Windows 2000 and later versions of Windows do not allow drivers to access hardware belonging to other driver stacks. A filter driver can be written to provide the functionality needed. If you wish to access bridge hardware, for instance, you must design a filter driver that implements the required operations on the bridge's configuration space. You must also provide an INF file that specifies the bridge hardware's possible hardware IDs, so the PnP manager can load the filter driver onto the bridge's driver stack when it detects the device ID of the bridge.

Alternatively, you can install a filter programmatically using **SetupDi*Xxx*** functions in the co-installer for your device.

The filter driver can then access the bridge using the **BUS_INTERFACE_STANDARD** interface.

Introduction to Controller Objects

6/25/2019 • 3 minutes to read • Edit Online

As its name suggests, a controller object usually represents a physical device controller with attached devices. A lowest-level non-WDM driver for a set of similar devices coordinated by a physical controller can create a controller object and use it to synchronize I/O operations between the attached devices. The driver implements a *ControllerControl* routine and calls the I/O manager's controller object support routines.

Note Use of controller objects is not supported in WDM drivers.

Generally, drivers use controller objects to synchronize operations to attached devices if the following criteria hold:

- The controller does not carry out long operations without interrupting, so the driver does not need to create a device-dedicated thread or use system worker threads.
- The devices connected to the controller are similar. That is, they are not devices with entirely different physical properties or operational functionality, such as the keyboard and mouse devices that can be connected to the keyboard and auxiliary device controller.
- The driver is designed to be monolithic: single-layered in relation to the device controller and attached physical devices, rather than being designed as a port driver (for the controller) with one or more class drivers (for attached devices) layered over the port driver.

Drivers of devices with I/O channels and a set of logical device objects also might use a controller object to synchronize their I/O operations between or among the channels of such a device.

A controller object has no name and thus is not the target of I/O requests. It is simply a synchronization mechanism to serialize I/O from a set of device objects. Because a controller object has no name, it is invisible to user-mode protected subsystems, which cannot make device I/O requests without getting a handle for the file object that represents the target device object. A controller object is also invisible to higher-level drivers, which cannot attach their own device objects to a controller object. In other words, neither the I/O manager nor a higher-level driver can set up an IRP requesting I/O on a device represented by a controller object. I/O requests are always issued to device objects. Only the driver can use the controller object.

Synchronization and Overlapped I/O

Monolithic drivers of physical devices with features like those of the "AT" disk controller are not required to use a controller object to synchronize their device I/O operations. For example, a driver writer could try something like the following synchronization technique instead of using a controller object:

- Set up named device objects to represent the devices that are targets for I/O requests.
- Maintain state information (perhaps a set of Device Busy flags in each device extension or in a single device extension) indicating which device object is the target of the current I/O operation.
- Carry out I/O operations for the currently busy device object and requeue incoming IRPs for other device objects until the current IRP is completed.

The preceding synchronization technique serializes IRP processing for all of the driver's target device objects. Note that it also forces the driver to complete the current IRP before its *StartIo* routine can begin processing the next IRP, which unfortunately decreases driver performance.

If certain device operations can be overlapped, using a controller object can increase a driver's I/O throughput, because this synchronization technique allows the driver to determine whether it can overlap operations just before it sets up the physical device. For example, a disk controller might allow the driver to overlap seeks on one disk

with read/write operations on another disk.

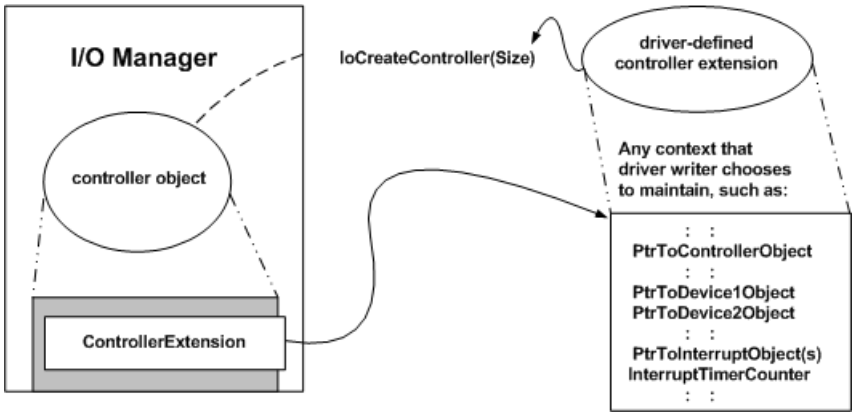
Moreover, using a controller object is a relatively easy way to synchronize I/O operations for more than one target device object through a single physical device, such as an "AT" disk controller. Using a controller object allows a monolithic driver to synchronize I/O operations across a set of named device objects without having to maintain state about every device and the device controller in one or more device extensions, and without having to requeue IRPs.

However, some devices that are designed to overlap I/O operations, such as full-duplex serial controllers or bus-master adapters, generally have drivers that set up internal queues for IRPs.

Creating Controller Objects and Controller Extensions

6/25/2019 • 2 minutes to read • Edit Online

If a driver uses a controller object, it must call **IoCreateController** after it has created device objects and its device is ready for I/O, typically after receiving a PnP **IRP_MN_START_DEVICE** request. The following figure illustrates the call.



Every controller object has an associated controller extension. As the previous figure shows, the caller of **IoCreateController** determines the *Size* of the controller extension. Its structure and contents are driver-defined.

In addition to whatever device-specific state information the driver maintains about the physical controller (or device with channels), the previous figure shows a representative set of driver-defined data for a controller extension.

The *PtrToControllerObject* pointer, returned by **IoCreateController**, must be passed in the driver's calls to **IoAllocateController** and **IoFreeController**, described in Allocating Controller Objects for I/O Operations. The driver must store the returned controller object pointer in the device extensions of its driver-created device objects or in another driver-accessible resident storage area (nonpaged pool, allocated by the driver). If the driver is unloaded, it also must pass the controller object pointer to **IoDeleteController**.

Most drivers that set up controller objects find it convenient to store a pointer to the current target device object or device extension in the controller extension. Usually, such a driver stores the controller object pointer in every one of its device extensions so that it can use the *ControllerObject*->**ControllerExtension** pointer to access driver-maintained, controller-specific state about I/O operations for every target device object.

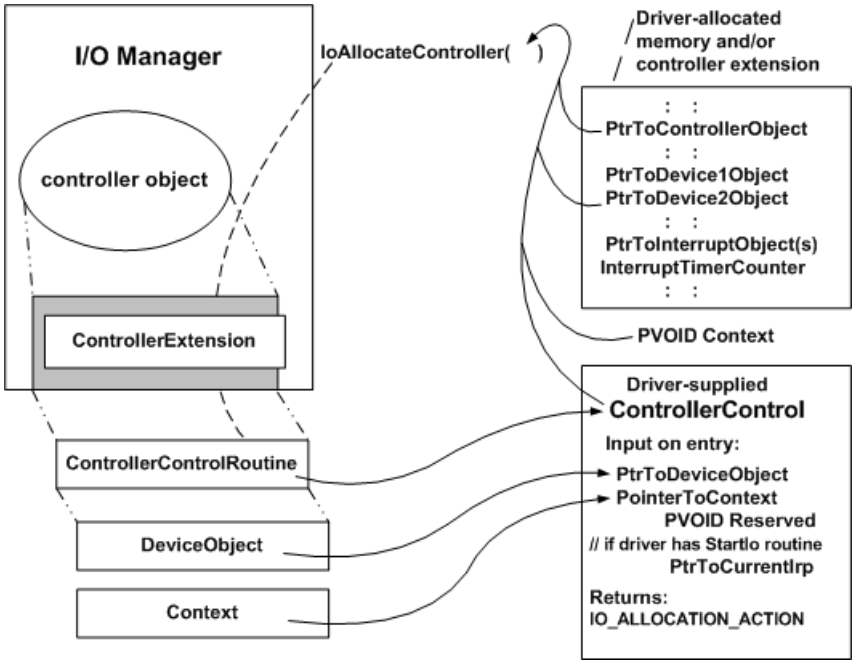
If the physical controller represented by a controller object generates interrupts, a driver also can use the controller extension as storage for *PtrToInterruptObject* pointers returned by **IoConnectInterrupt**. For more information, see Interrupt Service Routines.

IoCreateController allocates resident storage for the controller object and extension, which it initializes with zeros. If it cannot allocate the memory, **IoCreateController** returns a **NULL** pointer. If this occurs, the driver must fail device startup and should return STATUS_INSUFFICIENT_RESOURCES.

Allocating Controller Objects for I/O Operations

6/25/2019 • 3 minutes to read • Edit Online

After a driver that uses a controller object has started its device, it is ready to process IRPs sent to its target device objects. Whenever an IRP requires the driver to program the physical device represented by the controller object for an I/O operation, the driver calls **IoAllocateController**. The following figure illustrates such a call.



As the previous figure shows, a driver must supply more than the *ControllerObject* pointer that was returned by **IoCreateController** when it calls **IoAllocateController**. Along with this pointer, it must pass pointers to the device object representing the target of the current I/O request, to a driver-supplied *ControllerControl* routine, and to whatever *Context* its *ControllerControl* routine will need to set up the device for the requested I/O operation.

IoAllocateController queues the driver-supplied *ControllerControl* routine if the device represented by the controller object is busy. Otherwise, the *ControllerControl* routine is called immediately with the input parameters shown in the previous figure. The input *Context* pointer to **IoAllocateController** is passed to the driver's *ControllerControl* routine when it is run.

Use the following guidelines to determine where to store context information:

- The driver-supplied context area should not be in the controller extension unless the driver processes each IRP to completion before starting another operation on the physical controller. Otherwise, a context area in the controller extension could be overwritten by other driver routines or on receipt of a new IRP.
- Even if the driver overlaps a device I/O operation for another device object, a context area in the device extension of the target device object cannot be overwritten.
- If another I/O request is made for a particular device object and the driver has a *StartIo* routine, a context area in its device extension also cannot be overwritten because the incoming IRP will be queued when the driver calls **IoStartPacket** and the same IRP will remain in the device queue until the driver calls **IoStartNextPacket** just before it completes the current IRP for that device object.

The I/O manager passes a pointer to the *DeviceObject*->**CurrentIrp** to a *ControllerControl* routine if the driver has a *StartIo* routine. If a driver manages its own queuing of IRPs instead of having a *StartIo* routine, the I/O manager cannot give the *ControllerControl* routine a pointer to the current IRP. When the driver calls

IoAllocateController, it should pass the current IRP as part of the *Context*-accessible data.

The driver routine that calls **IoAllocateController** must be executing at IRQL = DISPATCH_LEVEL when the call occurs. A driver that makes this call from its *StartIo* routine is already running at DISPATCH_LEVEL.

The *ControllerControl* routine sets up the physical controller for the IRP's requested operation.

As shown in the previous figure, the *ControllerControl* routine returns a value of type **IO_ALLOCATION_ACTION**, which can be either of the following system-defined values:

- If the *ControllerControl* routine can start another operation on the physical controller, it should return **DeallocateObject** so the driver can overlap the next requested I/O operation.

  For example, if the *ControllerControl* routine can program a disk controller for a seek operation on one disk, complete that IRP, and return **DeallocateObject**, the *ControllerControl* routine can be called again to program the disk controller for a transfer operation on the other disk if any transfer requests currently are queued to the other disk.

- If the current IRP requires further processing by other driver routines, the *ControllerControl* routine must return **KeepObject**.

  For example, if the driver programs a disk controller for a transfer operation but cannot complete the IRP until the transfer is complete, the *ControllerControl* routine must return **KeepObject**.

When a *ControllerControl* routine returns **KeepObject**, usually the driver's ISR runs when the device interrupts, and its *DpcForIsr* or *CustomDpc* routine completes the I/O operation and the current IRP for the target device object.

Whenever the *ControllerControl* routine returns **KeepObject**, the routine that completes the IRP must call **IoFreeController**. Such a driver routine should call **IoFreeController** as soon as possible so that its next device I/O operation can be set up promptly.

Writing ControllerControl Routines

6/25/2019 • 2 minutes to read • Edit Online

Drivers that use a controller object must supply a *ControllerControl* routine to initiate I/O operations.

A lowest-level device driver that must synchronize operations through a physical controller, such as an "AT" disk controller, to similar devices can have a *ControllerControl* routine.

When a driver calls **IoAllocateController**, its *ControllerControl* routine is run immediately if the hardware represented by the controller object is available for an I/O operation. Otherwise, the *ControllerControl* routine is queued until the controller is free.

Note WDM drivers cannot use controller objects and *ControllerControl* routines.

Storage Requirements for ControllerControl Routines

6/25/2019 • 2 minutes to read • Edit Online

If it has a *ControllerControl* routine, a non-WDM driver must provide resident storage for a *ControllerObject* pointer returned by **IoCreateController**.

A driver can provide the necessary storage in a device extension or in nonpaged pool allocated by the driver. Usually, drivers that use controller objects store the *ControllerObject* pointer in the device extension of each device object that represents a physical or logical device controlled by the hardware represented by the controller object.

The driver writer determines the size and internal structure of the *ControllerObject*->**ControllerExtension**.

A controller object, which cannot be given a name, cannot be the target of an I/O request. The hardware it represents usually controls a set of homogeneous devices that are the actual targets of I/O requests.

Setting Up ControllerControl Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver's *DispatchPnP* routine must do the following when it receives an **IRP_MN_START_DEVICE** request, to set up a *ControllerControl* routine:

1. Call **IoCreateController** to set up the controller object, specifying the driver-determined *Size* for the controller extension, which the system allocates from nonpaged pool and initializes with zeros.
2. Save the *ControllerObject* pointer returned by **IoCreateController**, usually in the device extension of each device object representing a physical or logical device that is controlled by the hardware represented by the controller object.
3. Set up and/or initialize the driver-determined contents of the *ControllerObject*->**ControllerExtension**.

The returned *ControllerObject* pointer, the entry point of the driver's *ControllerControl* routine, the *DeviceObject* pointer representing the target device for the current IRP, and a *Context* pointer to an area already set up for the *ControllerControl* routine must be passed in the driver's calls to **IoAllocateController**. Usually, a driver's *StartIo* routine sets up the area at *Context* before it calls **IoAllocateController**.

ControllerControl Routine Requirements

6/25/2019 • 2 minutes to read • Edit Online

As its name implies, a *ControllerControl* routine is associated with a controller object. When the *ControllerControl* routine executes, the hardware represented by the controller object is free and the controller extension generally is not being accessed by another driver routine unless the controller extension contains context that is shared with the driver's ISR.

Usually, a *ControllerControl* routine does at least the following:

1. Updates or initializes whatever context the driver maintains in the device extension of the target device object and in the controller extension

   If the driver uses DMA, its *ControllerControl* routine usually is responsible for determining whether a given transfer request must be split up into partial transfers due to any system-imposed or device-imposed limitations on the size of each DMA transfer. In these circumstances, the *ControllerControl* routine also is responsible for calling **AllocateAdapterChannel** if the driver has an *AdapterControl* routine.

   If the driver uses PIO, its *ControllerControl* routine also is responsible for splitting transfer requests, if its hardware requires it, into partial-transfer ranges and for calling **MmGetSystemAddressForMdlSafe** with the MDL at **Irp->MdlAddress**.

2. Programs its hardware for the requested I/O operation

   If the device or controller extension can be accessed from the ISR, the *ControllerControl* routine must use a *SynchCritSection* routine that is invoked by calling **KeSynchronizeExecution**. For more information, see Using Critical Sections.

If the driver has a *Cancel* routine, its *ControllerControl* routine also must check the **Irp->Cancel** field to determine whether the current IRP should be canceled, and do either of the following:

If **Irp->Cancel** is set to **TRUE**, the *ControllerControl* routine must do the following:

1. Set STATUS_CANCELLED for **Status** and zero for **Information** in the I/O status block of the IRP.
2. Call **IoFreeController** to release the controller object so the next device operation can be started promptly.
3. Call **IoStartNextPacket** or dequeue the next IRP if the driver manages its own queuing.
4. Complete the canceled IRP with **IoCompleteRequest** and return control.

If **Irp->Cancel** is not set to **TRUE**, the *ControllerControl* routine instead must do the following:

1. Call **IoSetCancelRoutine** to reset the *Cancel* routine entry point for the IRP to **NULL**. Acquire the cancel spin lock for this call if the driver uses the I/O manager-supplied device queue in the device object.
2. Program the hardware for the requested I/O operation, using a *SynchCritSection* routine that is invoked by calling **KeSynchronizeExecution**. For more information, see Using Critical Sections

For more information about handling cancelable IRPs, see Canceling IRPs.

For most interrupt-driven I/O operations except overlapped operations on different devices attached to the physical controller/adapter, a *ControllerControl* routine should return **KeepObject** because the *DpcForIsr* or *CustomDpc* routine completes the operation and the IRP.

As soon as the I/O operation(s) to satisfy the current request are done, the routine that will complete the IRP

should call **IoFreeController** and **IoStartNextPacket** so that the next request can be processed as quickly as possible.

If the *ControllerControl* routine itself completes an IRP or if it can set up an operation, such as a disk seek, for one target device object (disk) that could be overlapped with an operation for another device object, the *ControllerControl* routine should return **DeallocateObject**.

Introduction to Interrupt Service Routines

6/25/2019 • 2 minutes to read • Edit Online

A driver of a physical device that receives interrupts registers one or more interrupt service routines (ISR) to service the interrupts. The system calls the ISR each time it receives that interrupt.

Devices for ports and buses prior to PCI 2.2 generate *line-based interrupts*. A device generates the interrupt by sending an electrical signal on a dedicated pin known as an *interrupt line*. Versions of Microsoft Windows prior to Windows Vista only support line-based interrupts.

Beginning with PCI 2.2, PCI devices can generate *message-signaled interrupts*. A device generates a message-signaled interrupt by writing a data value to a particular address. Windows Vista and later operating systems support both line-based and message-signaled interrupts.

The system supports two different types of ISRs:

- The driver can register an *InterruptService* routine to handle line-based or message-signaled interrupts. (This is the only type available prior to Windows Vista.) The system passes a driver-supplied context value.
- The driver can register an *InterruptMessageService* routine to handle message-signaled interrupts. The system passes both a driver-supplied context value and the message ID of the interrupt message.

For more information about registering an InterruptService or InterruptMessageService routine to service the device's interrupts, see Introduction to Message-Signaled Interrupts.

Removing an ISR

6/25/2019 • 2 minutes to read • Edit Online

Drivers can remove an ISR that is registered with **IoConnectInterruptEx** by calling **IoDisconnectInterruptEx**. **IoDisconectInterruptEx** takes a single *Parameters* parameter, which is a pointer to an **IO_DISCONNECT_INTERRUPT_PARAMETERS** structure. The values that are used for the members of the structure depend on the version that is used to register the ISR.

The driver must save certain information when it registers the ISR to later remove it. For the CONNECT_LINE_BASED and CONNECT_FULLY_SPECIFIED versions, the driver must save the value that is supplied in the **LineBased.InterruptObject** or **FullySpecified.InterruptObject** member of **IO_CONNECT_INTERRUPT_PARAMETERS**. For the CONNECT_MESSAGE_BASED version, the driver must save the values supplied in the **Version** and **MessageBased.ConnectionContext** members of **IO_CONNECT_INTERRUPT_PARAMETERS**.

Making an ISR Active or Inactive

6/25/2019 • 2 minutes to read • Edit Online

Starting with Windows 8, a driver can call the **IoReportInterruptActive** or **IoReportInterruptInactive** routine to make a registered interrupt service routine (ISR) active or inactive.

To register an ISR, and to connect the ISR to an interrupt or a set of interrupts, the driver calls the **IoConnectInterruptEx** routine. After the ISR is registered, the driver can use **IoReportInterruptActive** and **IoReportInterruptInactive** to perform lightweight (or "soft") connect and disconnect operations that leave the ISR's registration unchanged. **IoReportInterruptInactive** disables calls to the ISR by soft-disconnecting the associated interrupt or interrupts. **IoReportInterruptActive** soft-connects these interrupts to enable calls to the ISR.

For example, a driver might call **IoReportInterruptInactive** to soft-disconnect a set of interrupts before a device exits the D0 power state, and call **IoReportInterruptActive** to soft-connect these interrupts after the device reenters D0. In principle, a driver might instead call **IoDisconnectInterruptEx** before the device exits D0, and call **IoConnectInterruptEx** after the device reenters D0. However, **IoReportInterrupt*Xxx*** calls are faster than **IoConnectInterruptEx** and **IoDisconnectInterruptEx** calls. In contrast to **IoConnectInterruptEx** and **IoDisconnectInterruptEx** calls, which might fail for a variety of reasons (for example, insufficient system resources), **IoReportInterrupt*Xxx*** calls rarely, if ever, fail. Additionally, the**IoReportInterrupt*Xxx*** routines can be called at IRQL <= DISPATCH_LEVEL, whereas **IoConnectInterruptEx** and **IoDisconnectInterruptEx** can be called only at PASSIVE_LEVEL.

By default, the ISR is active (and calls to the ISR are enabled) after **IoConnectInterruptEx** successfully registers the ISR.

Calls to **IoReportInterruptInactive** and **IoReportInterruptActive** are optional. If a driver never calls these routines, the registered ISR stays active until the driver calls the **IoDisconnectInterruptEx** routine to unregister the ISR.

The driver should configure the device to issue interrupts only when the ISR for these interrupts is active. Failure to prevent a device from issuing interrupts when the ISR is inactive might cause system instability. For example, if a device shares a level-triggered interrupt line with other devices, and the device issues interrupt requests when the ISR is inactive, the ISRs for the other devices on the line will not acknowledge the interrupt and the interrupt will continue to fire. Before calling **IoReportInterruptInactive**, the driver should configure the device to stop issuing interrupts. After calling **IoReportInterruptActive**, the driver should configure the device to start issuing interrupts.

To unregister an ISR, a driver can call **IoDisconnectInterruptEx** regardless of whether the ISR is currently active or inactive.

An **IoReportInterruptActive** call that occurs when the ISR is already active has no effect, but is not treated as an error. Similarly, an **IoReportInterruptInactive** call that occurs when the ISR is already inactive has no effect, but is not treated as an error.

Interrupt Affinity

6/25/2019 • 2 minutes to read • Edit Online

The *affinity* of an interrupt is the set of processors that can service the interrupt. Each device has an *affinity policy*. The operating system uses the affinity policy to compute the affinity for that device's interrupts. The affinity policy can be specified in the device's INF file or registry settings.

Starting with Windows Vista, administrators can use the registry to set an affinity policy for an interrupt.

Administrators can set the following entries under the **\Interrupt Management\Affinity Policy** registry key:

- **DevicePolicy** is a REG_DWORD value that specifies an affinity policy. Each possible setting corresponds to a **IRQ_DEVICE_POLICY** value.
- **AssignmentSetOverride** is a REG_BINARY value that specifies a **KAFFINITY** mask. If **DevicePolicy** is 0x04 (**IrqPolicySpecifiedProcessors**), then this mask specifies a set of processors to assign the device's interrupts to.

The following table lists the **IRQ_DEVICE_POLICY** values, and the corresponding registry setting for **DevicePolicy**. For more information about the meaning of each value, see **IRQ_DEVICE_POLICY**.

IRQ_DEVICE_POLICY VALUE	NUMERIC VALUE IN REGISTRY
IrqPolicyMachineDefault	0x00
IrqPolicyAllCloseProcessors	0x01
IrqPolicyOneCloseProcessor	0x02
IrqPolicyAllProcessorsInMachine	0x03
IrqPolicySpecifiedProcessors	0x04
IrqPolicySpreadMessagesAcrossAllProcessors	0x05

A driver's INF file can provide default settings for the registry values. Here is an example of how to set the **DevicePolicy** value to **IrqPolicyOneCloseProcessor** in the INF file. For more information, see **INF AddReg Directive**.

```
[install-section-name.HW]
AddReg=add-registry-section

[add-registry-section]
HKR, "Interrupt Management\Affinity Policy", DevicePolicy, 0x00010001, 2
```

The system makes the registry settings available to the device's driver when it sends the **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** IRP to the driver. The operating system provides an **IO_RESOURCE_DESCRIPTOR** structure for each interrupt with the **Type** member set to

CmResourceTypeInterrupt. For a message-signaled interrupt, the CM_RESOURCE_INTERRUPT_MESSAGE bit of the **Flags** member is set; otherwise, it is clear. The **u.Interrupt** member describes the settings for the interrupt.

The following table gives the correspondence between registry settings and members of **u.Interrupt**.

REGISTRY VALUE	MEMBER OF U.INTERRUPT
DevicePolicy	**AffinityPolicy**
AssignmentSetOverride	**TargetedProcessors**

About KAFFINITY

The KAFFINITY type is an affinity mask that represents a set of logical processors in a group.

```
typedef ULONG_PTR  KAFFINITY;
```

The KAFFINITY type is 32 bits on a 32-bit version of Windows and is 64 bits on a 64-bit version of Windows.

If a group contains n logical processors, the processors are numbered from 0 to n-1. Processor number i in the group is represented by bit i in the affinity mask, where i is in the range 0 to n-1. Affinity mask bits that do not correspond to logical processors are always zero.

For example, if a KAFFINITY value identifies the active processors in a group, the mask bit for a processor is one if the processor is active, and is zero if the processor is not active.

The number of bits in the affinity mask determines the maximum number of logical processors in a group. For a 64-bit version of Windows, the maximum number of processors per group is 64. For a 32-bit version of Windows, the maximum number of processors per group is 32. Call the **KeQueryMaximumProcessorCountEx** routine to obtain the maximum number of processors per group. This number depends on the hardware configuration of the multiprocessor system, but can never exceed the fixed 64-processor and 32-processor limits that are set by the 64-bit and 32-bit versions of Windows, respectively.

The **GROUP_AFFINITY** structure contains an affinity mask and a group number. The group number identifies the group to which the affinity mask applies.

Kernel routines that use the KAFFINITY type include **IoConnectInterrupt**, **KeQueryActiveProcessorCount**, and **KeQueryActiveProcessors**.

Providing ISR Context Information

6/25/2019 • 2 minutes to read • Edit Online

On entry, an ISR receives a pointer to whatever context area the driver set up when it called **IoConnectInterruptEx** to register the routine.

Most drivers set the context pointer to the device object that represents the physical device that generates interrupts, or to that device object's device extension. In the device extension, the driver can store state information for the driver's ISR and *DpcForIsr* routine, the latter of which usually does almost all of the I/O processing to satisfy each request that caused the device to interrupt.

Typically, drivers use the device extension to store pointers to each of the device's interrupt objects (returned from calls to **IoConnectInterruptEx**). Drivers also typically store information in the device extension that allows an ISR to determine if an interrupt was issued by a device the ISR supports.

(Alternatively, interrupt object pointers can be stored in nonpaged pool that the driver allocates.)

Writing an ISR

6/25/2019 • 3 minutes to read • Edit Online

Drivers for physical devices that generate interrupts must have at least one interrupt service routine (ISR). The ISR must do whatever is appropriate to the device to dismiss the interrupt, possibly including stopping the device from interrupting. Then, it should do only what is necessary to save state and queue a DPC to finish the I/O operation at a lower priority (IRQL) than that at which the ISR executes.

A driver's ISR executes in an interrupt context, at some system-assigned *DIRQL*, as specified by the *SynchronizeIrql* parameter to **IoConnectInterruptEx**.

ISRs are interruptible. Another device with a higher system-assigned DIRQL can interrupt, or a high-IRQL system interrupt can occur, at any time.

Before the system calls an ISR, it acquires the interrupt's spin lock so the ISR cannot simultaneously execute on another processor. After the ISR returns, the system releases the spin lock.

Because an ISR runs at a relatively high IRQL, which masks off interrupts with an equivalent or lower IRQL on the current processor, it should return control as quickly as possible. Additionally, running an ISR at DIRQL restricts the set of support routines the ISR can call. For more information, see Managing Hardware Priorities.

Typically, an ISR performs the following general steps:

- If the device that caused the interrupt is not one supported by the ISR, the ISR immediately returns **FALSE**.
- Otherwise, the ISR clears the interrupt if necessary, saving whatever device context is necessary, and queues a DPC to complete the I/O operation at a lower IRQL. See DPC Objects and DPCs for more information. The ISR must then return **TRUE**.

Specifically, in drivers that do not overlap device I/O operations, the ISR should do the following:

1. Determine whether the interrupt is spurious. If so, return **FALSE** immediately so the ISR of the device that interrupted will be called promptly. Otherwise, continue interrupt processing.
2. Stop the device from interrupting, if necessary.
3. Gather whatever context information the *DpcForIsr* (or *CustomDpc*) routine will need to complete I/O processing for the current operation.
4. Store this context in an area accessible to the *DpcForIsr* or *CustomDpc* routine, usually in the device extension of the target device object for which processing the current I/O request caused the interrupt.

   If a driver overlaps I/O operations, the context information must include a count of outstanding requests the DPC routine is required to complete, along with whatever context the DPC routine needs to complete each request. If the ISR is called to handle another interrupt before the DPC has run, it must not overwrite the saved context for a request that has not yet been completed by the DPC.
5. If the driver has a *DpcForIsr* routine, call **IoRequestDpc** with pointers to the current IRP, the target device object, and the saved context. **IoRequestDpc** queues the *DpcForIsr* routine to be run as soon as IRQL falls below DISPATCH_LEVEL on a processor.

   If the driver has a *CustomDpc* routine, call **KeInsertQueueDpc** with a pointer to the DPC object (associated with the *CustomDpc* routine) and pointer(s) to any saved context the *CustomDpc* routine will need to complete the operation. Usually, the ISR also passes pointers to the current IRP and the target device object. The *CustomDpc* routine is run as soon as IRQL falls below DISPATCH_LEVEL on a processor.

6. Return **TRUE** to indicate that its device generated the interrupt.

In general, an ISR does no actual I/O processing to satisfy an IRP. Instead, it stops its device from interrupting, sets up necessary state information, and queues the driver's *DpcForIsr* or *CustomDpc* to do whatever I/O processing is necessary to satisfy the current request that caused the device to interrupt.

An ISR must run at DIRQL for the shortest possible interval. Following this guideline increases I/O throughput for every device in the machine because running at DIRQL masks off all interrupts to which the system has assigned a lesser or equal IRQL value.

The *SynchronizeIrql* of the driver's interrupt objects, specified when the driver called **IoConnectInterrupt**, determines the DIRQL at which the driver's ISR executes. For more information, see Synchronizing Access to Device Data.

Synchronizing Access to Device Data

10/8/2019 • 2 minutes to read • Edit Online

Typically, a driver's *InterruptService* or *InterruptMessageService* routines (ISRs) must share access to driver data and hardware resources with other driver routines. Since ISRs execute in an interrupt context at an elevated IRQL, and since a system might have multiple processors, it is important to synchronize access to shared data and resources so that each routine can be guaranteed to temporarily have exclusive access to this shared information, without interruption.

The system supports this synchronization by executing the ISR within an *interrupt critical section*. An interrupt has an assigned spin lock, the *interrupt spin lock*, and IRQL, the *interrupt synchronization IRQL*. The system guarantees this code executing within the critical section exclusive access to shared information, by raising the processor's IRQL to the interrupt synchronization IRQL and acquiring the interrupt spin lock before executing the code. The system always enters the interrupt's critical section before executing its ISR. Different interrupts can share the same critical section by sharing their interrupt spin lock and synchronization IRQL.

Drivers can implement code that runs in the interrupt's critical section by supplying a *SynchCritSection* routine. When the driver uses **KeSynchronizeExecution** to call the *SynchCritSection* routine, the system automatically enters the critical section for the interrupt specified by the *Interrupt* parameter.

Raising the processor's IRQL to the interrupt's synchronization IRQL prevents the current processor from being interrupted, except by an interrupt with a higher synchronization IRQL. Acquiring a spin lock prevents other processors from executing any critical section code associated with that spin lock.

The system assigns the interrupt spin lock and synchronization IRQL for the interrupt when the driver calls **IoConnectInterruptEx**. In most instances, the driver can allow the system to determine both values:

- If the driver uses the CONNECT_LINE_BASED version of **IoConnectInterruptEx**, and specifies a **NULL** spin lock, the system will allocate a spin lock for the interrupt line. The system also determines the value for the synchronization IRQL (drivers can optionally specify a higher value).
- If the driver uses the CONNECT_MESSAGE_BASED version of **IoConnectInterruptEx**, and specifies a **NULL** spin lock, the system will allocate a spin lock for each interrupt message. The system also determines the value of the synchronization IRQL for each message (drivers can optionally specify a higher value that will be common to all messages).

A driver must allocate its own spin lock only when using the CONNECT_FULLY_SPECIFIED version of **IoConnectInterruptEx** and when it has multiple interrupt vectors that must share the same critical section. A driver can specify its own spin lock and synchronization IRQL by using the **SpinLock** and **SynchronizeIrql** members of **IO_CONNECT_INTERRUPT_PARAMETERS**. For more information, see **IO_CONNECT_INTERRUPT_PARAMETERS**.

For information about writing and entering critical sections, see Using Critical Sections.

Registering an ISR

6/25/2019 • 2 minutes to read • Edit Online

Drivers use the **IoConnectInterruptEx** routine to register an ISR for an interrupt. **IoConnectInterruptEx** is part of Windows Vista and later operating systems. **IoConnectInterruptEx** takes a single *Parameters* parameter, which is a pointer to an **IO_CONNECT_INTERRUPT_PARAMETERS** structure. For Windows Server 2003, Windows XP, and Windows 2000, drivers can use the Iointex.lib library that is included in the Windows Driver Kit (WDK).

On Windows Vista and later, **IoConnectInterruptEx** provides several different methods for registering an ISR. The value specified for *Parameters*->**Version** determines the method, as follows:

- Use CONNECT_LINE_BASED to register an *InterruptService* routine for all of a device's line-based interrupts. (Devices usually have at most one line-based interrupt.) The system automatically detects any line-based interrupts assigned to the device. For more information, see Using the CONNECT_LINE_BASED Version of IoConnectInterruptEx.
- Use CONNECT_MESSAGE_BASED to register an *InterruptMessageService* routine for all of a device's message-signaled interrupts. You can also specify a fallback *InterruptService* routine—if the device only has line-based interrupts, **IoConnectInterruptEx** registers the *InterruptService* routine instead. The system automatically detects any message-signaled interrupts assigned to the device. For more information, see Using the CONNECT_MESSAGE_BASED Version of IoConnectInterruptEx.
- Use CONNECT_FULLY_SPECIFIED to register an *InterruptService* routine for each interrupt separately. You can use this to specify an *InterruptService* routine for either a line-based or a message-signaled interrupt, but you must manually specify the interrupt using information passed by the PnP manager. For more information, see Using the CONNECT_FULLY_SPECIFIED Version of IoConnectInterruptEx.

On operating systems prior to Windows Vista, you can only use CONNECT_FULLY_SPECIFIED. If you specify CONNECT_LINE_BASED or CONNECT_MESSAGE_BASED, **IoConnectInterruptEx** returns an error. You can use this behavior to determine if you are running on Windows Vista or an earlier system. For more information, see Using IoConnectInterruptEx Prior to Windows Vista.

Using the CONNECT_LINE_BASED Version of IoConnectInterruptEx

6/25/2019 • 2 minutes to read • Edit Online

For Windows Vista and later operating systems, a driver can use the CONNECT_LINE_BASED version of **IoConnectInterruptEx** to register an *InterruptService* routine for the driver's line-based interrupts. (Driver for earlier operating systems can use the CONNECT_FULLY_SPECIFIED version of **IoConnectInterruptEx**.)

Note You can use this method only for drivers that register a single interrupt service routine (ISR) for all of its line-based interrupts. If the driver can receive multiple interrupts, it must use the CONNECT_FULLY_SPECIFIED version of **IoConnectInterruptEx**.

The driver specifies a value of CONNECT_LINE_BASED for *Parameters*->**Version** and uses the members of *Parameters*->**LineBased** to specify the other parameters of the operation:

- *Parameters*->**LineBased.PhysicalDeviceObject** specifies the physical device object (PDO) for the device that the ISR services. The system uses the device object to automatically identify the device's line-based interrupts.
- *Parameters*->**LineBased.ServiceRoutine** points to the *InterruptService* routine, while *Parameters*->**LineBased**.**ServiceContext** specifies the value that the system passes as the *ServiceContext* parameter to *InterruptService*. The driver can use this to pass context information. For more information about passing context information, see Providing ISR Context Information.
- The driver provides a pointer to a PKINTERRUPT variable in *Parameters*->**LineBased.InterruptObject**. **IoConnectInterruptEx** sets this variable to point to the interrupt object for the interrupt, which can be used when removing the ISR. For more information, see Removing an ISR.
- Drivers can optionally specify a spin lock in *Parameters*->**LineBased.SpinLock** for the system to use when synchronizing with the ISR. Most drivers can just specify **NULL** to enable the system to allocate a spin lock on behalf of the driver. For more information about synchronizing with an ISR, see Synchronizing Access to Device Data.

The following code example demonstrates how to register an *InterruptService* routine using CONNECT_LINE_BASED:

```
IO_CONNECT_INTERRUPT_PARAMETERS params;

// deviceExtension is a pointer to the driver's device extension.
//     deviceExtension->IntObj is a PKINTERRUPT.
// deviceInterruptService is a pointer to the driver's InterruptService routine.
// PhysicalDeviceObject is a pointer to the device's PDO.
// ServiceContext is a pointer to driver-specified context for the ISR.

RtlZeroMemory( &params, sizeof(IO_CONNECT_INTERRUPT_PARAMETERS) );
params.Version = CONNECT_LINE_BASED;
params.LineBased.PhysicalDeviceObject = PhysicalDeviceObject;
params.LineBased.InterruptObject = &deviceExtension->IntObj;
params.LineBased.ServiceRoutine = deviceInterruptService;
params.LineBased.ServiceContext = ServiceContext;
params.LineBased.SpinLock = NULL;
params.LineBased.SynchronizeIrql = 0;
params.LineBased.FloatingSave = FALSE;

status = IoConnectInterruptEx(&params);

if (!NT_SUCCESS(status)) {
    // Operation failed. Handle error.
    ...
}
```

Using the CONNECT_MESSAGE_BASED Version of IoConnectInterruptEx

6/25/2019 • 2 minutes to read • Edit Online

For Windows Vista and later operating systems, a driver can use the CONNECT_MESSAGE_BASED version of **IoConnectInterruptEx** to register an ISR for the driver's message-signaled interrupts. The driver specifies a value of CONNECT_MESSAGE_BASED for *Parameters*->**Version**, and uses the members of *Parameters*->**MessageBased** to specify the other parameters of the operation.

- *Parameters*->**MessageBased.PhysicalDeviceObject** specifies the PDO for the device that the ISR services. The system uses the device object to automatically identify the device's message-signaled interrupts.
- *Parameters*->**MessageBased.MessageServiceRoutine** points to the *InterruptMessageService* routine, while *Parameters*->**MessageBased.ServiceContext** specifies the value that the system passes as the *ServiceContext* parameter to *InterruptMessageService*. The driver can use this to pass context information. For more information about passing context information, see Providing ISR Context Information.
- The driver can also specify a fallback *InterruptMessageService* routine in *Parameters*->**MessageBased.FallBackServiceRoutine**. If the device has line-based interrupts, but no message-signaled interrupts, the system will instead register the *InterruptMessageService* routine to service the line-based interrupts. In this case, the system passes *Parameters*->**MessageBased.ServiceContext** as the *ServiceContext* parameter to *InterruptService*. **IoConnectInterruptEx** updates *Parameters*->**Version** to CONNECT_LINE_BASED if it registered the fallback routine.
- *Parameters*->**MessageBased.ConnectionContext** points to a variable that receives a pointer to either a **IO_INTERRUPT_MESSAGE_INFO** (for *InterruptMessageService*) structure or a **KINTERRUPT** structure (for *InterruptService*). The driver can use the received pointer to remove the ISR. For more information, see Removing an ISR.
- Drivers can optionally specify a spin lock in *Parameters*->**MessageBased.SpinLock** for the system to use when synchronizing with the ISR. Most drivers can just specify **NULL** to enable the system to allocate a spin lock on behalf of the driver. For more information about synchronizing with an ISR, see Synchronizing Access to Device Data.

The following code example demonstrates how to register an *InterruptMessageService* routine by using CONNECT_MESSAGE_BASED.

```
IO_CONNECT_INTERRUPT_PARAMETERS params;

// deviceExtension is a pointer to the driver's device extension.
//     deviceExtension->IntInfo is a PVOID.
//     deviceExtension->IntType is a ULONG.
// deviceInterruptService is a pointer to the driver's InterruptService routine.
// deviceInterruptMessageService is a pointer to the driver's InterruptMessageService routine.
// PhysicalDeviceObject is a pointer to the device's PDO.
// ServiceContext is a pointer to driver-specified context for the ISR.

RtlZeroMemory( &params, sizeof(IO_CONNECT_INTERRUPT_PARAMETERS) );
params.Version = CONNECT_MESSAGE_BASED;
params.MessageBased.PhysicalDeviceObject = PhysicalDeviceObject;
params.MessageBased.MessageServiceRoutine = deviceInterruptMessageService;
params.MessageBased.ServiceContext = ServiceContext;
params.MessageBased.SpinLock = NULL;
params.MessageBased.SynchronizeIrql = 0;
params.MessageBased.FloatingSave = FALSE;
params.MessageBased.FallBackServiceRoutine = deviceInterruptService;

status = IoConnectInterruptEx(&params);

if (NT_SUCCESS(status)) {
    // We record the type of ISR registered.
    devExt->IsrType = params.Version;
} else {
    // Operation failed. Handle error.
    ...
}
```

Using the CONNECT_FULLY_SPECIFIED Version of IoConnectInterruptEx

6/25/2019 • 2 minutes to read • Edit Online

A driver can use the CONNECT_FULLY_SPECIFIED version of **IoConnectInterruptEx** to register an *InterruptService* routine for a specific interrupt. A driver can use the CONNECT_FULLY_SPECIFIED version starting with Windows Vista. By linking to the Iointex.lib library, the driver can use the CONNECT_FULLY_SPECIFIED version in Windows 2000, Windows XP, and Windows Server 2003. For more information, see Using IoConnectInterruptEx Prior to Windows Vista.

The driver specifies a value of CONNECT_FULLY_SPECIFIED for *Parameters*->**Version** and uses the members of *Parameters*->**FullySpecified** to specify the other parameters of the operation:

- *Parameters*->**FullySpecified.PhysicalDeviceObject** specifies the PDO for the device that the ISR services.
- *Parameters*->**FullySpecified.ServiceRoutine** points to the *InterruptService* routine, while *Parameters*->**FullySpecified.ServiceContext** specifies the value that the system passes as the *ServiceContext* parameter to *InterruptService*. The driver can use this to pass context information. For more information about passing context information, see Providing ISR Context Information.
- The driver provides a pointer to a PKINTERRUPT variable in *Parameters*->**FullySpecified.InterruptObject**. The **IoConnectInterruptEx** routine sets this variable to point to the interrupt object for the interrupt, which can be used when removing the ISR.
- Drivers can optionally specify a spin lock in *Parameters*->**FullySpecified.SpinLock** for the system to use when synchronizing with the ISR. Most drivers can just specify **NULL** to enable the system to allocate a spin lock on behalf of the driver. For more information about synchronizing with an ISR, see Synchronizing Access to Device Data.

The driver must specify the key properties of the interrupt in other members of *Parameters*->**FullySpecified**. The system provides the necessary information in the array of **CM_PARTIAL_RESOURCE_DESCRIPTOR** structures when it sends the **IRP_MN_START_DEVICE** IRP to the driver.

The system provides for each interrupt a **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure with **Type** member equal to **CmResourceTypeInterrupt**. For a message-signaled interrupt, the CM_RESOURCE_INTERRUPT_MESSAGE bit of the **Flags** member is set; otherwise, it is cleared.

The **u.Interrupt** member of **CM_PARTIAL_RESOURCE_DESCRIPTOR** contains the description of a line-based interrupt, while the **u.MessageInterrupt.Translated** member contains the description of a message-signaled interrupt. The following table indicates where, in the **CM_PARTIAL_RESOURCE_DESCRIPTOR** structure, to find the information required to set the members of *Parameters*->**FullySpecified** for both types of interrupt. For more information, see the code example that follows the table.

MEMBER	LINE-BASED INTERRUPT	MESSAGE-SIGNALED INTERRUPT
ShareVector	**ShareDisposition**	**ShareDisposition**
Vector	**u.Interrupt.Vector**	**u.MessageInterrupt.Translated.Vector**

MEMBER	LINE-BASED INTERRUPT	MESSAGE-SIGNALED INTERRUPT
Irql	**u.Interrupt.Level**	**u.MessageInterrupt.Translated.Level**
InterruptMode	**Flags** & CM_RESOURCE_INTERRUPT_LATCHED	**Flags** & CM_RESOURCE_INTERRUPT_LATCHED
ProcessorEnableMask	**u.Interrupt.Affinity**	**u.MessageInterrupt.Translated.Affinity**

A driver will only receive **CM_PARTIAL_RESOURCE_DESCRIPTOR** structures for message-signaled interrupts on Windows Vista and later versions of Windows.

The following code example demonstrates how to register an *InterruptService* routine using CONNECT_FULLY_SPECIFIED.

```
IO_CONNECT_INTERRUPT_PARAMETERS params;

// deviceExtension is a pointer to the driver's device extension.
//     deviceExtension->IntObj is a PKINTERRUPT.
// deviceInterruptService is a pointer to the driver's InterruptService routine.
// IntResource is a CM_PARTIAL_RESOURCE_DESCRIPTOR structure of either type CmResourceTypeInterrupt or
CmResourceTypeMessageInterrupt.
// PhysicalDeviceObject is a pointer to the device's PDO.
// ServiceContext is a pointer to driver-specified context for the ISR.

RtlZeroMemory( &params, sizeof(IO_CONNECT_INTERRUPT_PARAMETERS) );
params.Version = CONNECT_FULLY_SPECIFIED;
params.FullySpecified.PhysicalDeviceObject = PhysicalDeviceObject;
params.FullySpecified.InterruptObject = &devExt->IntObj;
params.FullySpecified.ServiceRoutine = deviceInterruptService;
params.FullySpecified.ServiceContext = ServiceContext;
params.FullySpecified.FloatingSave = FALSE;
params.FullySpecified.SpinLock = NULL;

if (IntResource->Flags & CM_RESOURCE_INTERRUPT_MESSAGE) {
    // The resource is for a message-signaled interrupt. Use the u.MessageInterrupt.Translated member of
IntResource.

    params.FullySpecified.Vector = IntResource->u.MessageInterrupt.Translated.Vector;
    params.FullySpecified.Irql = (KIRQL)IntResource->u.MessageInterrupt.Translated.Level;
    params.FullySpecified.SynchronizeIrql = (KIRQL)IntResource->u.MessageInterrupt.Translated.Level;
    params.FullySpecified.ProcessorEnableMask = IntResource->u.MessageInterrupt.Translated.Affinity;
} else {
    // The resource is for a line-based interrupt. Use the u.Interrupt member of IntResource.

    params.FullySpecified.Vector = IntResource->u.Interrupt.Vector;
    params.FullySpecified.Irql = (KIRQL)IntResource->u.Interrupt.Level;
    params.FullySpecified.SynchronizeIrql = (KIRQL)IntResource->u.Interrupt.Level;
    params.FullySpecified.ProcessorEnableMask = IntResource->u.Interrupt.Affinity;
}

params.FullySpecified.InterruptMode = (IntResource->Flags & CM_RESOURCE_INTERRUPT_LATCHED ? Latched :
LevelSensitive);
params.FullySpecified.ShareVector = (BOOLEAN)(IntResource->ShareDisposition == CmResourceShareShared);

status = IoConnectInterruptEx(&params);

if (!NT_SUCCESS(status)) {
    ...
}
```

Using Passive-Level Interrupt Service Routines

6/25/2019 • 4 minutes to read • Edit Online

Starting with Windows 8, a driver can use the **IoConnectInterruptEx** routine to register a passive-level *InterruptService* routine (ISR). When the associated interrupt occurs, the kernel's interrupt trap handler schedules this routine to run at IRQL = PASSIVE_LEVEL. An ISR might need to run at passive level if it can access the hardware registers of a device only through I/O requests. A passive-level ISR can synchrononously send an I/O request to a device and block until the request completes.

Registering a Passive-Level ISR

The input parameter to **IoConnectInterruptEx** is a pointer to an **IO_CONNECT_INTERRUPT_PARAMETERS** structure. To register a passive-level ISR, set the **Version** member of this structure to either CONNECT_FULLY_SPECIFIED or CONNECT_LINE_BASED. If **Version** = CONNECT_FULLY_SPECIFIED, set the **Irql** member to PASSIVE_LEVEL, the **SynchronizeIrql** member to PASSIVE_LEVEL, and the **SpinLock** member to **NULL**. If **Version** = CONNECT_LINE_BASED, set **SynchronizeIrql** = PASSIVE_LEVEL and **SpinLock** = **NULL**.

If the interrupt object specifies a passive-level ISR, the **KeSynchronizeExecution** routine uses a kernel synchronization event object instead of a spin lock to synchronize execution of the *SynchCritSection* routine with the ISR.

This event object is allocated by the **IoConnectInterruptEx** routine in the call that registers the passive-level ISR. The caller must not supply a spin lock in this call. (That is, the caller must set the **SpinLock** member of the **IO_CONNECT_INTERRUPT_PARAMETERS** structure to NULL if the ISR is to run at passive level.) Otherwise, **IoConnectInterruptEx** fails and returns error status STATUS_INVALID_PARAMETER.

The **KeAcquireInterruptSpinLock** and **KeReleaseInterruptSpinLock** routines cause a bug check if the ISR for the supplied interrupt object runs at IRQL = PASSIVE_LEVEL.

Devices that Require Passive-Level Interrupt Handling

For a memory-mapped device that signals a level-triggered interrupt request, the device's ISR is typically called at DIRQL from within the kernel's interrupt trap handler. The ISR manipulates the hardware registers in the device to turn off the interrupt.

However, an ISR might need to run at IRQL = PASSIVE_LEVEL if the associated device signals a level-triggered interrupt request but the device's hardware registers cannot be accessed directly from an ISR that is called at DIRQL from within the kernel's interrupt trap handler. For example, the device registers might not be memory-mapped, or the ISR might be temporarily blocked during a register access.

Starting with Windows 8, a driver can register a passive-level ISR. When the interrupt occurs, the kernel's interrupt trap handler schedules the ISR to run at IRQL = PASSIVE_LEVEL. Before the handler returns, it must silence the interrupt in the interrupt controller (or GPIO controller). If a device signals an edge-triggered interrupt, the handler clears the interrupt in the interrupt controller. If the device signals a level-triggered interrupt, the handler temporarily masks the interrupt in the interrupt controller; after the ISR runs, the kernel unmasks the interrupt.

An Example

An example of a device that might require a passive-level ISR is a sensor device that is connected to a low-power serial bus, such as I^2C. Starting with Windows 8, support for I^2C and for other simple peripheral buses (SPBs) is

provided by the SPB framework extension (SpbCx).

To access the registers of the I²C-connected sensor device, the sensor driver sends the sensor device an I/O request, which is jointly handled by SpbCx and by the controller driver for the bus. To perform the requested operation, the SPB controller must transfer data serially over the bus. This transfer is relatively slow and cannot be performed within the time constraints of an ISR that runs at DIRQL. However, a passive-level ISR can send the I/O request synchronously and then block until the request completes.

The passive-level ISR in this example might be blocked for a longer time if the I²C bus controller is turned off when the ISR sends the I/O request to the interrupting device. In this case, the controller must complete the transition to the D0 power state before it can transfer the data over the bus.

In contrast to a bus such as PCI, the I²C bus in this example provides no bus-specific means to convey interrupt requests from peripheral devices to the processor. Instead, the sensor device might signal an interrupt to a pin on a GPIO controller device, which then relays the interrupt request to the processor. For more information, see GPIO Interrupts.

Typically, the hardware registers of a GPIO controller are memory-mapped and can be accessed at DIRQL by the kernel's interrupt trap handler. When the sensor device causes an interrupt, the handler must silence the interrupt by manipulating the interrupt bits in the GPIO controller's registers.

For a level-triggered interrupt, the kernel's interrupt trap handler masks the interrupt request at the GPIO pin, and then schedules the sensor device's ISR to run at passive level. The ISR must clear the interrupt request from the sensor device. After the ISR returns, the kernel unmasks the interrupt request at the GPIO pin.

For an edge-triggered interrupt, the kernel's trap handler clears the interrupt request at the GPIO pin, and then schedules the sensor device's ISR to run at passive level.

Worker Routines

In the call to **IoConnectInterruptEx**, a driver has the option to split the processing of the interrupt between a passive-level ISR and a worker routine. As a general rule, the ISR should do the initial processing of the interrupt (for example, silence a level-triggered interrupt), and defer additional processing to the worker. Although both the ISR and worker run at passive level, the ISR runs at a relatively high priority and might delay other high-priority tasks. These tasks might include passive-level ISRs for new interrupts.

In rare cases, an interrupt might require so little processing that the passive-level ISR can perform all of the processing for the interrupt, and no worker routine is required.

For information about using passive-level ISRs in KMDF drivers, see Supporting Passive-Level Interrupts.

Using IoConnectInterruptEx Prior to Windows Vista

6/25/2019 • 2 minutes to read • Edit Online

A driver for Windows 2000, Windows XP, or Windows Server 2003 can link to the Iointex.lib library to use **IoConnectInterruptEx** on those versions of the operating system.

To use **IoConnectInterruptEx** in such a driver, include Iointex.h in the source code for your driver, immediately following Wdm.h or Ntddk.h. The Iointex.h header declares a prototype for the routine. When you build your driver, make sure that it is statically linked to Iointex.lib.

For operating systems prior to Windows Vista, the version of **IoConnectInterruptEx** provided by Iointex.lib only supports the CONNECT_FULLY_SPECIFIED version of the routine. If any other version is specified, the routine returns an NTSTATUS error code, and sets *Parameters*->**Version** to CONNECT_FULLY_SPECIFIED.

Using this behavior, you can write your driver so that it uses CONNECT_LINE_BASED or CONNECT_MESSAGE_BASED on Windows Vista, and CONNECT_FULLY_SPECIFIED on earlier operating systems. First call **IoConnectInterruptEx** with *Parameters*->**Version** equal to CONNECT_LINE_BASED or CONNECT_MESSAGE_BASED. If the return value is an error code and *Parameters*->**Version** != CONNECT_FULLY_SPECIFIED, then retry the operation with *Parameters*->**Version** set to CONNECT_FULLY_SPECIFIED.

The following code example illustrates the technique:

```
IO_CONNECT_INTERRUPT_PARAMETERS params;

// deviceExtension is a pointer to the driver's device extension.
//     deviceExtension->MessageUsed is a BOOLEAN.

RtlZeroMemory( &params, sizeof(IO_CONNECT_INTERRUPT_PARAMETERS) );
params.Version = CONNECT_MESSAGE_BASED;

// Set members of params.MessageBased here.

status = IoConnectInterruptEx(&params);

if ( NT_SUCCESS(status) ) {
    // Operation succeeded. We are running on Windows Vista.
    devExt->MessageUsed = TRUE; // We save this for posterity.
} else {
    // Check to see if we are running on an operating system prior to Windows Vista.
    if (params.Version == CONNECT_FULLY_SPECIFIED) {
        devExt->MessageUsed = FALSE;  // We're not using message-signaled interrupts.

        // Set members of params.FullySpecified here.

        status = IoConnectInterruptEx(&params);
    } else {
        // Other error.
    }
}
```

Introduction to Message-Signaled Interrupts

6/25/2019 • 2 minutes to read • Edit Online

Message-signaled interrupts (MSIs) were introduced in the PCI 2.2 specification as an alternative to line-based interrupts. Instead of using a dedicated pin to trigger interrupts, devices that use MSIs trigger an interrupt by writing a value to a particular memory address. PCI 3.0 defines an extended form of MSI, called *MSI-X*, that enables greater programmability. Windows Vista and later versions of Windows support MSI and MSI-X. A single device can support both MSI and MSI-X. For such a device, the operating system will automatically use MSI-X.

An *interrupt message* is a particular value that a device writes to a particular address to trigger an interrupt. Unlike line-based interrupts, message-signaled interrupts have edge semantics. The device sends a message but does not receive any hardware acknowledgment that the interrupt was received.

For PCI 2.2, a message consists of an address and a partially opaque 16-bit value. Each device is assigned a single address. To send multiple messages, the device can use the lower 4 bits of the message value to distinguish messages. Therefore, for PCI 2.2, devices can support up to 16 messages.

For PCI 3.0, a message consists of an address and an opaque 32-bit value. Each different message has its own unique address. Unlike for PCI 2.2, the device does not modify the value. For PCI 3.0, a device can support up to 2,048 different messages. Devices that support PCI 3.0 MSI-X feature a dynamically programmable hardware table that contains entries for each of the interrupt sources in the device. Each entry in this table can be programmed with one of the messages that are allocated to a device, and can be independently masked. Drivers can change the programming of an interrupt message into a table entry and whether an entry has been masked. For more information, see Dynamically Configuring MSI-X.

Drivers can register a single *InterruptMessageService* routine that handles all possible messages or individual *InterruptService* routines for each message.

Drivers can handle MSIs that a device sends as follows:

1. During driver installation, enable MSIs in the registry. You can also use the registry to specify the number of messages to allocate for the device. For more information, see Enabling Message-Signaled Interrupts in the Registry.

2. Optionally, increase the number of interrupt messages and save some per-message settings by responding to an **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** request. For more information, see Using Interrupt Resource Descriptors.

3. In the driver's dispatch routine for **IRP_MN_START_DEVICE**, call **IoConnectInterruptEx** to register an *InterruptService* or *InterruptMessageService* routine to service the device's interrupts. Use the CONNECT_FULLY_SPECIFIED version of **IoConnectInterruptEx** to register an *InterruptService* routine for a specific message or the CONNECT_MESSAGE_BASED version of **IoConnectInterruptEx** to register a single *InterruptMessageService* routine for all messages. For more information, see Using the CONNECT_MESSAGE_BASED Version of IoConnectInterruptEx and Using the CONNECT_FULLY_SPECIFIED Version of IoConnectInterruptEx.

4. After the driver no longer intends to service interrupts from the device, call **IoDisconnectInterruptEx** (after disabling the device's interrupts) to remove any registered interrupt service routines.

Drivers that are designed to use multiple messages should check that the expected number of messages is allocated. If the Plug and Play (PnP) manager cannot allocate the requested number of messages, it instead allocates exactly one message to the device. Drivers can check the number of messages that are actually allocated in one of the following ways:

- The PnP manager reports the number of allocated messages in its list of raw resource descriptors. For more information, see Using Interrupt Resource Descriptors.
- When **IoConnectInterruptEx** returns, it sets *Parameters*->**MessageBased.ConnectContext.InterruptMessageTable->MessageCount** to the number of allocated messages.

Enabling Message-Signaled Interrupts in the Registry

6/25/2019 • 2 minutes to read • Edit Online

To receive message-signaled interrupts (MSIs), a driver's INF file must enable MSIs in the registry during installation. Use the **Interrupt Management\MessageSignaledInterruptProperties** subkey of the device's hardware key to enable MSI support.

The **MSISupported** entry of **Interrupt Management\MessageSignaledInterruptProperties** is a REG_DWORD value that determines whether the device supports MSIs. Set **MSISupported** to 1 to enable MSI support.

You can also use the registry to specify the maximum number of MSIs to allocate for their device. The **MessageNumberLimit** entry of **Interrupt Management\MessageSignaledInterruptProperties** is a REG_DWORD value that specifies the maximum number of MSIs to allocate. For PCI 2.2, **MessageNumberLimit** must be 1, 2, 4, 8, or 16. For PCI 3.0, **MessageNumberLimit** can be any number up to 2,048.

Use an **INF AddReg Directive** in your driver's INF file to set registry keys under the device's hardware key. For more information, see **INF DDInstall.HW Section**.

The following code example shows how to set the **MSISupported** entry under **Interrupt Management\MessageSignaledInterruptProperties** for the device. Note that you must first create the **Interrupt Management** and **Interrupt Management\MessageSignaledInterruptProperties** keys before you can set the **MSISupported** entry.

```
[mydevice.HW]
AddReg = mydevice_addreg

[mydevice_addreg]
HKR,Interrupt Management,,0x00000010
HKR,Interrupt Management\MessageSignaledInterruptProperties,,0x00000010
HKR,Interrupt Management\MessageSignaledInterruptProperties,MSISupported,0x00010001,1
```

Using Interrupt Resource Descriptors

6/25/2019 • 3 minutes to read • Edit Online

The Plug and Play (PnP) manager assigns interrupt messages to a device using two passes. First, the PnP manager sends an **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** request to the driver with a list of hardware resources, including interrupt messages, that it intends to assign to the device. The driver can modify this list to change the number of interrupt messages, as well as some per-message settings. Then, after the PnP manager actually assigns the resources, it sends an **IRP_MN_START_DEVICE** request and supplies a complete list of the hardware resources, including interrupt messages, assigned to the driver's device.

The **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** request supplies a list of **IO_RESOURCE_DESCRIPTOR** structures. If the device has an MSI (message-signaled interrupt) capability structure as defined in the PCI 2.2 specification, the PnP manager supplies a single interrupt message descriptor. If the device has an MSI-X capability structure as defined in the PCI 3.0 specification, the PnP manager supplies one structure for each interrupt message. Interrupt message descriptors have **Type** = **CmResourceTypeInterrupt** and **Flags** = CM_RESOURCE_INTERRUPT_LATCHED | CM_RESOURCE_INTERRUPT_MESSAGE. Drivers can also change settings such as the interrupt affinity by changing the **u.Interrupt** members of the structure. Note that when using MSI, interrupts all have same affinity, while when using MSI-X they can have different affinities. For more information, see Interrupt Affinity and Priority.

For MSI, as defined in PCI 2.2, **u.Interrupt.MaximumVector** - **u.Interrupt.MinimumVector** + 1 is the number of interrupt messages allocated for the device. Drivers can change the number of interrupt messages by modifying **u.Interrupt.MinimumVector**. For MSI interrupt messages, **u.Interrupt.MaximumVector** is always CM_RESOURCE_INTERRUPT_MESSAGE_TOKEN. To allocate *MessageCount* interrupt messages, set **u.Interrupt.MinimumVector** to equal CM_RESOURCE_INTERRUPT_MESSAGE_TOKEN - *MessageCount* + 1.

For MSI-X, as defined in PCI 3.0, drivers can change the number of interrupt messages allocated by adding or removing entries from the list. Note that interrupt message resources added this way must not be subsequently removed in response to the **IRP_MN_START_DEVICE** request. For MSI-X, the PnP manager supplies one descriptor per message interrupt, and the **u.Interrupt.MinimumVector** and **u.Interrupt.MaximumVector** members of this descriptor are both set to CM_RESOURCE_INTERRUPT_MESSAGE_TOKEN.

Once the Plug and Play manager has assigned all hardware resources for the device, including interrupt messages, it sends the **IRP_MN_START_DEVICE** request to the driver. This request supplies two lists of **CM_PARTIAL_RESOURCE_DESCRIPTOR** structures, one each for raw and translated resources. For interrupt messages, the PnP manager supplies one structure for each allocated memory address with **Type** = **CmResourceTypeInterrupt** and **Flags** = CM_RESOURCE_INTERRUPT_LATCHED | CM_RESOURCE_INTERRUPT_MESSAGE.

Note that when using MSI, the driver only receives one interrupt resource descriptor, since all messages share the same address. The **MessageCount** member of **u.MessageInterrupt.Raw** can be used to determine the number of messages assigned. When using MSI-X, the driver receives a separate resource descriptor for each interrupt message.

In Windows 8, the operating system does not support resource requests for more than 2048 interrupt messages per device function. In Windows 7 and Windows Vista, the operating system does not support resource requests for more than 910 interrupt messages per device function. If the device driver exceeds this limit, the device might fail to start. To enable a driver to operate in a computer that contains many logical processors, the driver should avoid requesting more than one interrupt per processor.

During system rebalancing of interrupt resources, the PnP manager might ask a driver to select a preferred set of

alternate interrupt resources from a resource requirements list. However, the PnP manager cannot always assign to a driver the resources that the driver prefers. The driver must therefore tolerate, without failures, the assignment of any set of alternate interrupt resources from the resource requirements list. For example, the device might be assigned a smaller number of message interrupts than the driver requested. In the worst case, the driver must be prepared to operate the device with just one line-based interrupt.

Dynamically Configuring MSI-X

6/25/2019 • 2 minutes to read • Edit Online

Windows Vista Service Pack 1 (SP1), Windows Server 2008, and later operating systems support dynamically modifying the properties of MSI-X interrupt messages. (The PCI 3.0 specification defined MSI-X.) The PCI bus driver exposes the GUID_MSIX_TABLE_CONFIG_INTERFACE interface to allow drivers for PCI devices to modify the settings in the bus hardware interrupt table.

Drivers use the interface by sending an **IRP_MN_QUERY_INTERFACE** request to the bus driver, with the *InterfaceType* parameter equal to GUID_MSIX_TABLE_CONFIG_INTERFACE. The bus driver supplies a pointer to a **PCI_MSIX_TABLE_CONFIG_INTERFACE** structure, which supplies pointers to three routines that modify the interrupt table:

- *SetTableEntry* assigns a message ID to the hardware table entry.
- *MaskTableEntry* masks the interrupt corresponding to a hardware table entry.
- *UnmaskTableEntry* unmasks the interrupt corresponding to a hardware table entry.

By default, the interrupt table is configured so that the first entry has message ID zero, the second entry has message ID one, and so on. If the number of table entries exceeds the number of messages, each additional table entry is assigned message ID zero. (The message ID is the index for the interrupt's entry in the **MessageInfo** member of the **IO_INTERRUPT_MESSAGE_INFO** structure that describes the driver's message-signaled interrupts. The **IoConnectInterruptEx** routine supplies a pointer to this structure.)

Introduction to DPC Objects

6/25/2019 • 2 minutes to read • Edit Online

Because ISRs must execute as quickly as possible, drivers must usually postpone the completion of servicing an interrupt until after the ISR returns. Therefore, the system provides support for *deferred procedure calls* (DPCs), which can be queued from ISRs and which are executed at a later time and at a lower IRQL than the ISR.

Each DPC is associated with a system-defined *DPC object*. The system supplies one DPC object for each device object. The system initializes this DPC object when a driver registers a DPC routine known as the *DpcForIsr* routine. A driver can create additional DPC objects if more than one DPC is needed. These extra DPCs are known as *CustomDpc* routines.

DPC object contents should not be directly referenced by drivers. The object's structure is not documented. Drivers do not have access to the system-supplied DPC object assigned to each device object. Drivers allocate storage for extra DPCs, but the contents of these DPC objects should only be referenced by system routines.

DPC objects and DPCs can also be used with timers. For more information, see Timer Objects and DPCs.

Introduction to DPCs

6/25/2019 • 2 minutes to read • Edit Online

Any driver that has an ISR typically also has at least one *DpcForIsr* or *CustomDpc* routine to complete processing of interrupt-driven I/O operations. A typical lowest-level driver's *DpcForIsr* or *CustomDpc* routine does the following:

- Finishes handling an I/O operation that the ISR began processing.
- Dequeues the next IRP so that the driver can begin processing it.
- Completes the current IRP, if possible.

Sometimes the current IRP cannot be completed because several data transfers are required, or a recoverable error was detected. In these cases, the *DpcForIsr* or *CustomDpc* routine typically reprograms the device for either another transfer or a retry of the last operation.

A *DpcForIsr* or *CustomDpc* routine is called in an arbitrary DPC context at IRQL DISPATCH_LEVEL. Running at DISPATCH_LEVEL restricts the set of support routines a *DpcForIsr* or *CustomDpc* routine can call. See Managing Hardware Priorities for more information.

DPC objects and DPCs can also be used with timers. For more information, see Timer Objects and DPCs.

Which Type of DPC Should You Use?

6/25/2019 • 2 minutes to read • Edit Online

Depending on a driver's design, it can have any of the following:

- A single *DpcForIsr* to complete all interrupt-driven I/O operations
- A set of one or more *CustomDpc* routines.
- Both a *DpcForIsr* and a set of operation-specific *CustomDpc* routines

Whether a driver has a single *DpcForIsr* routine, a set of *CustomDpc* routines, or both, depends on the nature of its underlying device and the set of I/O requests it must support.

Most lowest-level device drivers have a single *DpcForIsr* routine to complete I/O processing for each IRP that requires one or more operations on their respective devices. Using a single *DpcForIsr* to complete per-request, interrupt-driven I/O operations on a device that does one operation at a time is relatively easy. Such a driver's ISR need only call **IoRequestDpc** for each interrupt-driven I/O operation.

Few lowest-level drivers have *CustomDpc* routines unless their devices require more than one DPC to complete a varied set of interrupt-driven I/O operations.

Using a single *DpcForIsr* to complete overlapped, interrupt-driven I/O operations on a device that can do concurrent operations is possible with careful design, but can be relatively difficult. In addition to or instead of queuing a *DpcForIsr*, an ISR can queue a set of operation-specific, driver-supplied *CustomDpc* routines by calling **KeInsertQueueDpc**.

For example, consider some of the design challenges involved in writing a serial driver. As the driver of a full-duplex device, a serial driver cannot rely on a one-to-one correspondence between the order in which IRPs are queued to a *StartIo* routine and the sequence of interrupts from its device in a multitasking, multiprocessor system. Furthermore, serial drivers must handle timing out requests and asynchronous user-generated requests to cancel previously requested operations, to purge buffered data, and so forth.

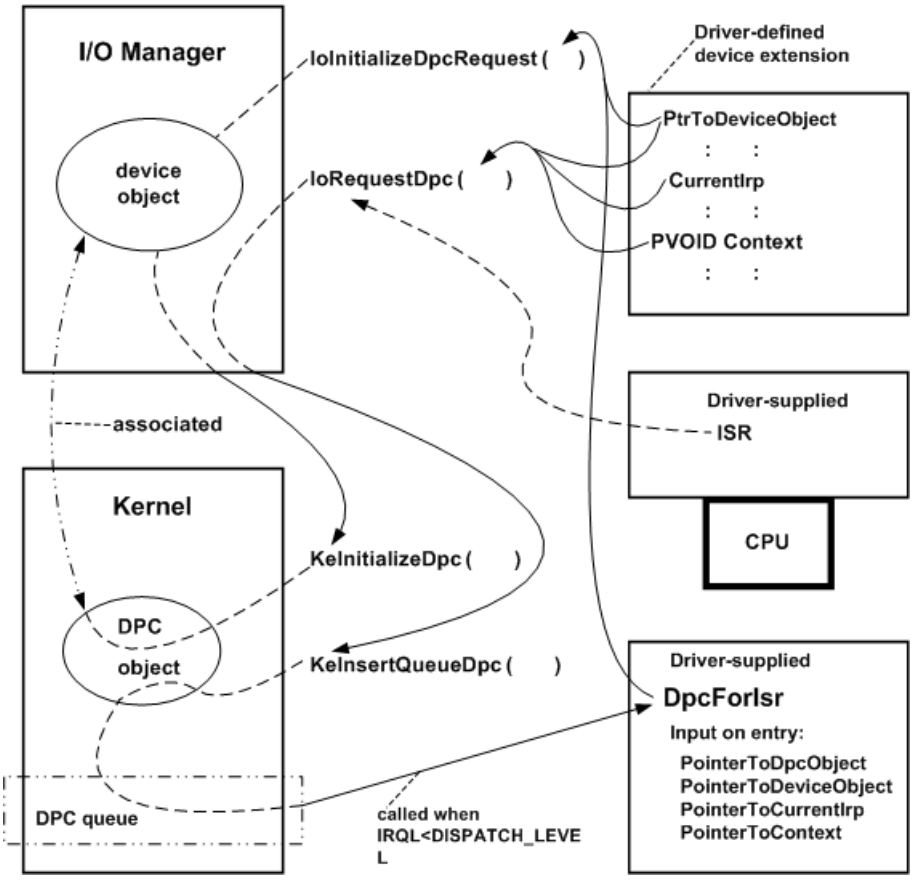
Consequently, a serial driver might maintain internal queues for the read, write, purge, and wait operations that user-mode COM port applications can request. It also could maintain reference counts or use some other tracking mechanism, such as a set of flags, for the IRPs in its internal queues. Its ISR would call **KeInsertQueueDpc** with any of a number of driver-allocated and initialized DPC objects, each associated with a driver-supplied *CustomDpc* routine.

Registering and Queuing a DpcForIsr Routine

6/25/2019 • 2 minutes to read • Edit Online

A driver registers a *DpcForIsr* routine for a device object by calling **IoInitializeDpcRequest** after it has created the device object. The driver can make this call from its *AddDevice* routine, or from *DispatchPnP* code that handles **IRP_MN_START_DEVICE** requests.

To queue the *DpcForIsr* routine for execution, the driver's ISR calls **IoRequestDpc** just before it returns. The following figure illustrates calls to these routines.



As the previous figure shows, calling **IoInitializeDpcRequest** associates a DPC object with a driver-supplied *DpcForIsr* routine and a driver-created device object. The I/O manager allocates memory for the DPC object and calls **KeInitializeDpc** on the driver's behalf.

When the ISR is called to handle a device interrupt at DIRQL, it should return control to the system as soon as possible for better overall system and driver performance. Usually, an ISR merely clears the interrupt, gathers whatever context information the *DpcForIsr* routine needs to complete the operation that caused the interrupt, calls **IoRequestDpc**, and returns.

When the ISR calls **IoRequestDpc**, it passes a pointer to the device object, a pointer to the *DeviceObject*->**CurrentIrp**, and a pointer to a driver-determined context. The I/O manager calls **KeInsertQueueDpc** on the driver's behalf, which queues the DPC object. When IRQL falls below DISPATCH_LEVEL on a processor, the kernel dequeues the DPC object and runs the driver's *DpcForIsr* routine on that processor at IRQL DISPATCH_LEVEL.

The *DpcForIsr* routine is responsible for doing whatever is necessary to complete the I/O requested in the current IRP. On entry, the routine receives a pointer to the DPC object, along with pointers to the device object, IRP, and context area, which were passed in the ISR's call to **IoRequestDpc**. The context area must be in resident memory, and is usually in the device extension. Alternatively, it can be in nonpaged pool allocated by the driver, or in a

controller extension if the driver uses a controller object.

Because ISR and *DpcForIsr* routines can run concurrently on SMP machines, you must follow these guidelines:

- The ISR must call **IoRequestDpc** just before it returns control. Otherwise, the *DpcForIsr* routine might be run on another processor before the ISR has finished setting up the context area for the *DpcForIsr* routine.
- The *DpcForIsr* routine and any other driver routine that shares a context area with the ISR must call **KeSynchronizeExecution**, specifying a driver-supplied *SynchCritSection* routine that accesses the shared context area in a multiprocessor-safe manner.
- If a driver uses the device extension to maintain context about its device I/O operations, the *DpcForIsr* routine should never call **IoStartNextPacket** for the input device object (nor dequeue an IRP for the input device object, if the driver manages its own IRP queuing) until just before it calls **IoCompleteRequest**.

  Otherwise, the driver's *StartIo* (or queue-management routines) might start another I/O operation that overwrites the shared context area before the *DpcForIsr* routine can complete the current operation. This is because the ISR can be called again if the device interrupts while or before the *DpcForIsr* routine executes (assuming interrupts are still enabled).

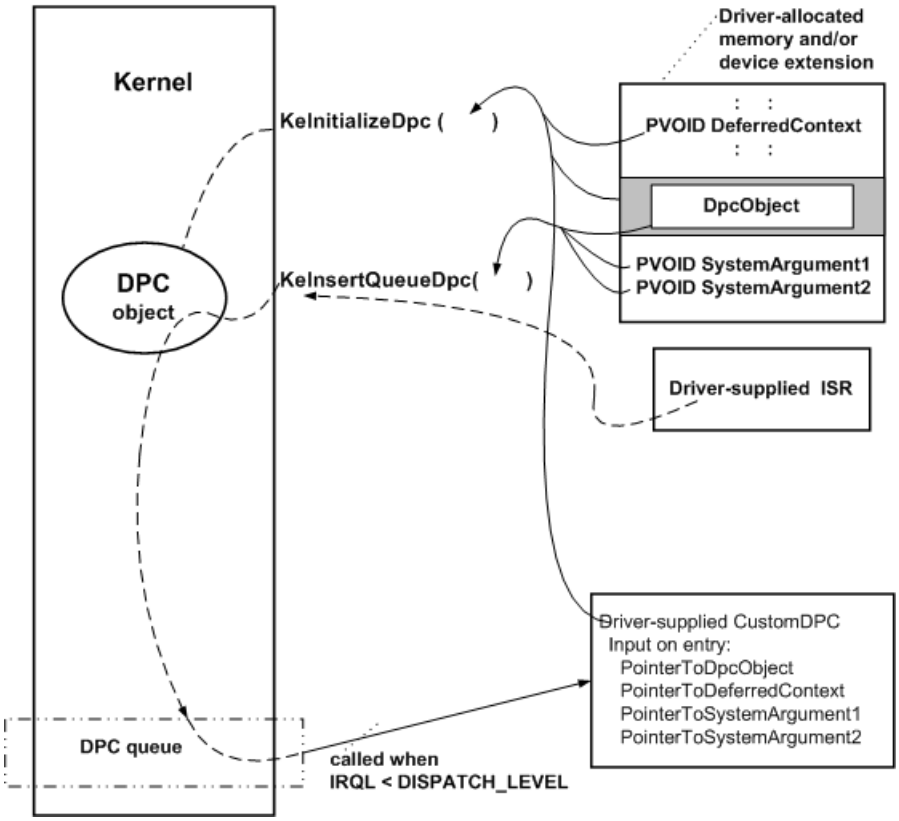
Even on a uniprocessor machine, the ISR could be called again if the device interrupts while or before the *DpcForIsr* routine is run. If this occurs, the *DpcForIsr* routine is run only once. In other words, there is no one-to-one correspondence between an ISR's calls to **IoRequestDpc** and instantiations of the *DpcForIsr* routine if a driver overlaps I/O operations for its target device objects.

Registering and Queuing a CustomDpc Routine

6/25/2019 • 2 minutes to read • Edit Online

A driver registers a *CustomDpc* routine for a device object by calling **KeInitializeDpc** after it has created the device object. The driver can make this call from its *AddDevice* routine, or from *DispatchPnP* code that handles **IRP_MN_START_DEVICE** requests.

Just before the driver's ISR returns control, it can call **KeInsertQueueDpc** to queue the *CustomDpc* routine for execution. The following figure illustrates calls to these routines.



As the previous figure shows, a driver that has a *CustomDpc* routine must provide the storage for a DPC object. Because the driver must pass a pointer to the DPC object from its ISR, the storage must be in resident, system-space memory. Most drivers with *CustomDpc* routines provide storage for their DPC objects in the device extension, but the storage can be in a controller extension if the driver uses a controller object or in nonpaged pool allocated by the driver.

When the driver calls **KeInitializeDpc**, it must pass the entry point for its *CustomDpc* routine, along with pointers to the driver-allocated storage for the DPC object and to a driver-defined context area, which is passed to the *CustomDpc* routine when it is called. Because the context area must be accessible at IRQL = DISPATCH_LEVEL, it also must be in resident memory.

Unlike a *DpcForIsr* routine, a *CustomDpc* routine is not associated with a device object. Nevertheless, drivers typically include pointers to the target device object and current IRP in the context information supplied to the *CustomDpc* routine. Like a *DpcForIsr* routine, the *CustomDpc* routine uses this information to complete an interrupt-driven I/O operation at a lower IRQL than the ISR.

As the previous figure shows, the ISR passes pointers to the DPC object and to two additional parameters, which are driver-defined, to **KeInsertQueueDpc**. If all processors in the machine currently have code running at an IRQL greater than or equal to DISPATCH_LEVEL, the DPC object is queued until the IRQL falls below DISPATCH_LEVEL

on a processor. Then, the kernel dequeues the DPC object and the driver's *CustomDpc* routine is run on the processor at IRQL DISPATCH_LEVEL.

Only a single instantiation of any one DPC object can be queued at any given moment. Thus if an ISR calls **KeInsertQueueDpc** more than once with the same *Dpc* pointer before the driver's *CustomDpc* routine is run, the *CustomDpc* routine runs only once after IRQL falls below DISPATCH_LEVEL on a processor.

A *CustomDpc* routine is responsible for doing whatever is necessary to complete the I/O operation that caused the interrupt.

The ISR and *CustomDpc* routines can be run concurrently on an SMP machine. Therefore, when writing *CustomDpc* routines, follow the guidelines set out in the previous section, Registering and Queuing a DpcForIsr Routine.

Handling Overlapped I/O Operations

6/25/2019 • 2 minutes to read • Edit Online

The *DpcForIsr* or *CustomDpc* routine of a driver that overlaps operations on its device cannot rely on a one-to-one correspondence between requests input to the *StartIo* routine and the ISR's calls to **IoRequestDpc** or **KeInsertQueueDpc**. Such a driver's *DpcForIsr* or *CustomDpc* cannot necessarily use the input pointers to the IRP and ISR-supplied context, or the **CurrentIrp** pointer in the target device object, to complete only that IRP.

At any given moment, the same DPC object cannot be queued twice. If an ISR calls **IoRequestDpc** or **KeInsertQueueDpc** more than once before the corresponding *DpcForIsr* or *CustomDpc* executes, the DPC routine runs only once when the IRQL on a processor falls below DISPATCH_LEVEL. On the other hand, if the ISR calls **IoRequestDpc** or **KeInsertQueueDpc** while the corresponding *DpcForIsr* or *CustomDpc* is running on another processor, the DPC routine can run on two processors concurrently.

Therefore, any driver that overlaps interrupt-driven I/O operations on its device must have the following:

- A *DpcForIsr* or *CustomDpc* routine that can complete some driver-maintained count of outstanding requests each time it is called
- An ISR that never overwrites the context information that it passes to a *DpcForIsr* or *CustomDpc* routine, until that routine has used the context information and completed the IRP to which the context information belongs
- A *SynchCritSection* routine that accesses the ISR's context area on behalf of the *DpcForIsr* or *CustomDpc* routine

Writing DPC Routines

6/25/2019 • 2 minutes to read • Edit Online

The primary responsibilities of *DpcForIsr* and *CustomDpc* routines are ensuring that the next device I/O operation is started promptly and completing the current IRP.

Additional work done by any *DpcForIsr* or *CustomDpc* routine depends on the driver's design and the nature of the device. For example, a *DpcForIsr* or *CustomDpc* routine also can do any of the following:

- Retry an operation that has timed out or failed.
- Call **IoAllocateErrorLogEntry**, set up an error log packet to report a device I/O error, and call **IoWriteErrorLogEntry**.

  For more information about handling I/O errors, see Logging Errors.
- If the driver uses buffered I/O, or if the IRP specifies a device control operation, transfer data read in from the device to the system buffer at **Irp->AssociatedIrp.SystemBuffer** before completing the IRP.
- If the driver uses direct I/O and must break large transfers into smaller pieces, save state about each just-completed partial-transfer operation, calculate the next partial-transfer range, and use a driver-supplied *SynchCritSection* routine to program the device for the next partial-transfer operation.

  Even a driver that uses buffered I/O might have to split up a transfer request if its device has limited transfer capabilities.
- If the driver uses packet-based DMA, call **FlushAdapterBuffers** after each device transfer operation, and call **FreeAdapterChannel** or **FreeMapRegisters** when a sequence of partial transfers is done and the full transfer request is satisfied.

  If a requested transfer is only partly satisfied by a single DMA operation, the *DpcForIsr* or *CustomDpc* routine is usually responsible for setting up one or more DMA operations until the IRP's specified number of bytes have been fully transferred.

  For more information about using DMA, see Adapter Objects and DMA.
- If the driver uses programmed I/O (PIO), call **KeFlushIoBuffers** at the end of each transfer operation if the current IRP requests a read.

  If a requested transfer is only partly satisfied by a single PIO operation, the *DpcForIsr* or *CustomDpc* routine is usually responsible for setting up one or more transfer operations until the IRP's specified number of bytes have been fully transferred.

  For more information about using PIO, see Using Direct I/O.
- If a non-WDM driver has a *ControllerControl* routine, call **IoFreeController** when a requested operation is complete.

Note that a *DpcForIsr* or *CustomDpc* routine usually does most of the driver's device I/O processing to satisfy IRPs. These routines also share some of the responsibility for queuing IRPs to the device with the driver's dispatch routines.

Consider the following a general design guidelines.

- Any *DpcForIsr* or *CustomDpc* routine should call **IoStartNextPacket** as soon as it can safely make this call: that is, without possibly causing a resource conflict or race condition with the driver's *StartIo* routine or with

any other routine the *StartIo* routine causes to run.

- If a driver manages its own queuing of IRPs, its *DpcForIsr* or *CustomDpc* routine should notify the driver as soon as it is safe to dequeue the next IRP and to set up the device for the next request.

A *DpcForIsr* or *CustomDpc* routine must call **IoStartNextPacket**, or otherwise notify the appropriate driver routine when device I/O processing for the next request can be started. Depending on the driver and its device, this can occur well before the *DpcForIsr* or *CustomDpc* routine completes the current IRP with **IoCompleteRequest**, or it can occur immediately before this routine completes the current IRP and returns control.

Guidelines for Writing DPC Routines

6/25/2019 • 3 minutes to read • Edit Online

Keep the following points in mind when writing a *DpcForIsr* or *CustomDpc* routine:

- A *DpcForIsr* or *CustomDpc* routine must synchronize its access to a physical device, and to any shared state information or resources that the driver maintains, with the driver's other routines that access the same device or memory locations.

  If a *DpcForIsr* or *CustomDpc* routine shares the device or state with an ISR, it must call **KeSynchronizeExecution**, supplying the address of a driver-supplied *SynchCritSection* routine that programs the device or accesses the shared state. For more information, see Using Critical Sections.

  If a *DpcForIsr* or *CustomDpc* routine shares state or resources, such as an interlocked queue or a timer object, with routines other than an ISR, it must protect the shared state or resources with a driver-initialized executive spin lock. For more information, see Spin Locks.

- *DpcForIsr* and *CustomDpc* routines run at IRQL = DISPATCH_LEVEL, which restricts the set of support routines they can call.

  For example, *DpcForIsr* and *CustomDpc* routines can neither access nor allocate pageable memory, and they cannot wait for kernel dispatcher objects to be set to the signaled state. On the other hand, they can acquire and release a driver's executive spin lock with **KeAcquireSpinLockAtDpcLevel** and **KeReleaseSpinLockFromDpcLevel**, which run faster than **KeAcquireSpinLock** and **KeReleaseSpinLock**.

  Although a DPC routine cannot make blocking calls, it can queue a work item to run in a system worker thread that runs at IRQL = PASSIVE_LEVEL. The work item can make blocking calls that wait on dispatcher objects. To queue a work item, a *DpcForIsr* routine typically calls a routine such as **IoQueueWorkItem**, and a *CustomDpc* routine typically calls the **ExQueueWorkItem** routine.

- *DpcForIsr* and *CustomDpc* routines are typically responsible for starting the next I/O operation on the device.

  For lowest-level physical device drivers that use direct I/O, this responsibility can include using a *SynchCritSection* routine to program the device to transfer more data in order to satisfy the current IRP before the driver calls **IoStartNextPacket**.

- *DpcForIsr* and *CustomDpc* routines should run only for brief periods, and should delegate as much processing as possible to worker threads.

  While a DPC routine runs on a processor, all threads are prevented from running on the same processor. Other DPC routines that are queued and ready to run can be blocked from executing until the current DPC routine is finished. To avoid degrading system responsiveness, a typical DPC routine should run for no more than 100 microseconds each time it is called. If a task requires longer than 100 microseconds and must execute at IRQL = DISPATCH_LEVEL, the DPC routine should end after 100 microseconds and schedule one or more *CustomTimerDpc* routines to complete the task at a later time. For more information about *CustomTimerDpc* routines, see Timer Objects and DPCs.

  A DPC routine should perform only tasks that must run at DISPATCH_LEVEL, and then delegate any remaining interrupt-related work to threads that run at IRQL = PASSIVE_LEVEL. For example, a DPC routine can queue a work item to run in a system worker thread.

  DPC routines that call the **KeStallExecutionProcessor** routine to delay execution must not specify delays

of more than 100 microseconds.

Use the performance analysis tools in the WDK to evaluate the execution times of DPC routines. For an example that uses the Tracelog tool to monitor DPC execution times, see Example 15: Measuring DPC/ISR Time.

- If the driver uses DMA and its *AdapterControl* routine returns **KeepObject** or **DeallocateObjectKeepRegisters** (thereby retaining the system DMA controller channel or bus-master adapter for additional transfer operations), the *DpcForIsr* or *CustomDpc* routine is responsible for releasing the adapter object or map registers with **FreeAdapterChannel** or **FreeMapRegisters** before it completes the current IRP and returns control.
- If a lowest-level physical device driver sets up a controller object to synchronize I/O operations through the controller to attached devices, its *DpcForIsr* or *CustomDpc* routine is responsible for releasing the controller object using **IoFreeController** before it completes the current IRP and returns control.
- *DpcForIsr* and *CustomDpc* routines are generally responsible for logging any device errors that occurred during the processing of a given request, retrying the current request if necessary and possible, and for setting the I/O status block and calling **IoCompleteRequest** for the current IRP.
- If the driver and device support overlapped I/O operations, the driver must follow the rules for handling overlapped I/O operations.
- The *DpcForIsr* or *CustomDpc* routine of any driver usually completes the I/O processing only for a subset of the public I/O control codes that the driver must support. In particular, the DPC routine completes operations for device control requests with the following characteristics:
  - Requests that change the state of the physical device
  - Requests that require the return of inherently volatile information about the physical device

Organization of DPC Queues

6/25/2019 • 2 minutes to read • Edit Online

The system provides one DPC queue for each processor. Drivers can control which queue the system assigns a DPC to, the location of the DPC within the queue, and when the queue is processed.

DPCs that are assigned to a particular processor's queue are run on that processor. By default, when the driver calls **KeInsertQueueDpc** or **IoRequestDpc**, the DPC is queued on the currently active processor. Drivers can specify the processor queue by calling **KeSetTargetProcessorDpc** before calling **KeInsertQueueDpc** or **IoRequestDpc**.

On Windows Vista and later versions of Windows, the system also has one threaded DPC queue for each processor. Drivers can use **KeSetTargetProcessorDpc** to specify the processor queue for threaded DPCs.

The **KeSetImportanceDpc** routine controls where a DPC is placed within the queue. Typically, the DPC is placed at the end of the queue; but if the driver first calls **KeSetImportanceDpc** with the *Importance* parameter equal to **HighImportance**, the DPC is placed at the beginning of the queue.

For ordinary (non-threaded) DPCs, **KeSetImportanceDpc** also determines whether **KeInsertQueueDpc** or **IoRequestDpc** will immediately begin processing the DPC queue. The following list describes the rules for processing the queue:

- Processing of the DPC queue begins immediately if the DPC is assigned to the current processor and *Importance* is not equal to **LowImportance**, or if *Importance* is equal to **LowImportance** and the DPC queue depth of the current processor exceeds a system-defined limit or the DPC request rate has fallen below a system-defined minimum. Otherwise, processing of the DPC is deferred until the appropriate queue depth and rate requirements are met.
- Processing of the DPC queue begins immediately on the target processor if the DPC is assigned to a processor that is different than the current processor and *Importance* equals **MediumHighImportance** or **HighImportance**, or if the DPC queue depth of the target processor exceeds a system-defined limit. Otherwise, processing of the DPC is deferred until the appropriate queue depth and rate requirements are met.

Introduction to Threaded DPCs

6/25/2019 • 2 minutes to read • Edit Online

Threaded DPCs are available in Windows Vista and later versions of Windows.

A *threaded DPC* is a DPC that the system executes at IRQL = PASSIVE_LEVEL. Threaded DPCs are enabled by default, but you can disable them by setting the **HKLM\System\CCS\Control\SessionManager\Kernel\ThreadDpcEnable** registry key to zero. When threaded DPCs are disabled, they execute as ordinary DPCs.

An ordinary DPC preempts the execution of all threads, and cannot be preempted by a thread or by another DPC. If the system has a large number of ordinary DPCs queued, or if one of those DPCs runs for a long time, every thread will remain paused for an arbitrarily long time. Thus, each ordinary DPC increases system latency, which can hurt the performance of time-sensitive applications, such as audio or video playback.

Conversely, a threaded DPC can be preempted by an ordinary DPC, but not by other threads. Therefore, you should use threaded DPCs rather than ordinary DPCs—unless a particular DPC must not be preempted, not even by another DPC.

The system represents threaded DPCs (and ordinary DPCs) as **KDPC** structures. To initialize a **KDPC** structure for a threaded DPC, call the **KeInitializeThreadedDpc** routine, and pass it a *CustomThreadedDpc* routine that performs the action of the DPC.

Because a *CustomThreadedDpc* routine can execute at either PASSIVE_LEVEL or DISPATCH_LEVEL, you must ensure that your *CustomThreadedDpc* routine correctly synchronizes at both IRQLs. For more information about how to do so, see Synchronization and Threaded DPCs.

In addition, you must ensure that your *CustomThreadedDpc* routine obeys all the restrictions for DISPATCH_LEVEL code. If threaded DPCs are enabled, they run at IRQL = PASSIVE_LEVEL but are still subject to the same restrictions as ordinary DPCs. All of the code that executes in a threaded DPC—including all functions that are called by the *CustomThreadedDpc* routine—must conform to the restrictions of the DPC environment. For example, code must not block on passive-level synchronization objects, such as KEVENT objects. Most existing device stacks, such as networking, storage, and USB, do not support threaded DPC processing, and they might try to block if they detect that they are called at PASSIVE_LEVEL. For similar reasons, the Kernel-Mode Driver Framework (KMDF) does not support threaded DPC processing, and KMDF drivers should not try to use threaded DPCs. For more information about the DPC environment, see Writing DPC Routines.

To add a threaded DPC to the DPC queue, call **KeInsertQueueDpc**. To remove a threaded DPC from the queue before it executes, call **KeRemoveQueueDpc**.

Synchronization and Threaded DPCs

6/25/2019 • 2 minutes to read • Edit Online

To synchronize access to a memory location that is accessed from both inside and outside a *CustomThreadedDpc* routine, a driver can use ordinary spin locks or queued spin locks. When doing so, the driver must obey certain rules to correctly synchronize at IRQL= PASSIVE_LEVEL and at IRQL = DISPATCH_LEVEL, because a *CustomThreadedDpc* routine can execute at both IRQLs.

For an ordinary spin lock, the following rules apply:

- To acquire and release the spin lock, the driver can call **KeAcquireSpinLock** and **KeReleaseSpinLock** from both inside and outside the *CustomThreadedDpc* routine.
- The driver can call **KeAcquireSpinLockForDpc** and **KeReleaseSpinLockForDpc** from inside the *CustomThreadedDpc* routine. Note that the *CustomThreadedDpc* routine must not call **KeAcquireSpinLockAtDpcLevel** or **KeReleaseSpinLockFromDpcLevel**, because these routines can safely be called only at IRQL = DISPATCH_LEVEL.

The rules for queued spin locks are similar:

- To acquire and release the spin lock, the driver can call **KeAcquireInStackQueuedSpinLock** and **KeReleaseInStackQueuedSpinLock** from both inside and outside the *CustomThreadedDpc* routine.
- The driver can call **KeAcquireInStackQueuedSpinLockForDpc** and **KeReleaseInStackQueuedSpinLockForDpc** from inside the *CustomThreadedDpc* routine. Note that the *CustomThreadedDpc* routine must not call **KeAcquireInStackQueuedSpinLockAtDpcLevel** or **KeReleaseInStackQueuedSpinLockFromDpcLevel**, because these routines can safely be called only at IRQL = DISPATCH_LEVEL.

Because **KeAcquireSpinLockForDpc** and **KeAcquireInStackQueuedSpinLockForDpc** do not reset the IRQL when called at DISPATCH_LEVEL, they execute faster than **KeAcquireSpinLock** and **KeAcquireInStackQueuedSpinLock**, respectively.

For more information about spin locks, see Spin Locks.

Converting an Ordinary DPC to a Threaded DPC

6/25/2019 • 2 minutes to read • Edit Online

Converting an ordinary DPC to a threaded DPC is straightforward. Simply replace the call to **KeInitializeDpc** (which initializes the DPC) with one to **KeInitializeThreadedDpc**, and refer to the following table to replace the calls inside the DPC routine that acquire and release spin locks.

ORDINARY DPC CALL	CORRESPONDING THREADED DPC CALL
KeAcquireSpinLockAtDpcLevel	**KeAcquireSpinLockForDpc**
KeReleaseSpinLockFromDpcLevel	**KeReleaseSpinLockForDpc**
KeAcquireInStackQueuedSpinLockAtDpcLevel	**KeAcquireInStackQueuedSpinLockForDpc**
KeReleaseInStackQueuedSpinLockFromDpcLevel	**KeReleaseInStackQueuedSpinLockForDpc**

You do not need to change calls to other spin lock routines, such as **KeAcquireSpinLock** or **KeAcquireInStackQueuedSpinLock**.

Using Critical Sections

6/25/2019 • 2 minutes to read • Edit Online

Any driver that contains an *InterruptService* routine will most likely require one or more critical sections to synchronize access to hardware resources or driver data among the ISR and other routines.

This section includes the following topics:

Introduction to SynchCritSection Routines

6/25/2019 • 2 minutes to read • Edit Online

Critical sections are sections of code that require exclusive access to hardware resources or driver data. That is, the code must not be interrupted by other code that can reference the same resources or data, and the resources or data must not be referenced by more than one processor at a time.

Critical sections should be confined to ISRs and *SynchCritSection* value and acquiring a spin lock. After a *SynchCritSection* routine returns, the system releases the spin lock and lowers the processor's IRQL.

Raising the processor's IRQL to the device's DIRQL value prevents the current processor from being interrupted, except by a higher-priority device. Acquiring a spin lock prevents other processors from executing any critical section code associated with that spin lock. (This spin lock is sometimes called an *interrupt spin lock*.)

A device driver's *StartIo* and *DpcForIsr* or *CustomDpc* routines frequently must access some of the same hardware resources (such as device registers or other bus-relative memory) or driver-maintained data as the driver's ISR. Depending on the driver's device or design, its dispatch, *AdapterControl*, *ControllerControl*, or timer routines also might access hardware resources or driver-maintained data.

To call any non-ISR critical section, a driver must use the **KeSynchronizeExecution** routine. This routine accepts the address of a *SynchCritSection* routine as input, along with driver-defined context information and an interrupt object pointer. The system uses the interrupt object pointer to determine the DIRQL and spin lock to use with the *SynchCritSection* routine. (The driver previously supplied these values, using the **IoConnectInterrupt** function's *SpinLock* and *SynchronizeIrql* parameters.)

Writing SynchCritSection Routines

6/25/2019 • 2 minutes to read • Edit Online

Drivers use their *SynchCritSection* routines for either of two basic purposes:

Programming a device for an I/O operation

Accessing shared state information

Like an ISR, a *SynchCritSection* routine must execute as quickly as possible, doing only what is necessary to set up device registers or update context data, before returning.

Because **KeSynchronizeExecution** holds a device driver's interrupt spin lock while its *SynchCritSection* routine runs, the driver's ISR cannot execute until the *SynchCritSection* routine returns control.

For any received IRP, a device driver should do as much I/O processing as possible either at IRQL PASSIVE_LEVEL in its dispatch routines (or possibly device-dedicated threads), or at IRQL DISPATCH_LEVEL in its *StartIo* routine and DPC routines.

For additional information about how critical sections are synchronized, see Using Spin Locks: An Example.

Programming a Device for an I/O Operation

6/25/2019 • 2 minutes to read • Edit Online

Use the following general guidelines for designing, writing, and calling *SynchCritSection* routines that program a device for I/O operations:

- A *SynchCritSection* routine that programs the device for I/O operations must return control as quickly as possible.

  For this reason, the *SynchCritSection* routine should do only what is necessary to set up the device for I/O. Therefore, the driver should perform all IRP preprocessing, initializing state information for other driver routines, and acquiring hardware resources before it calls the *SynchCritSection* routine.

- A device driver can have multiple *SynchCritSection* routines to program the device.

  For example, the driver of a device for which setting up a read request differs markedly from setting up certain device control requests might have separate *SynchCritSection* routines to program its device for each type of request.

- Every *SynchCritSection* routine must return control as quickly as possible, because running any *SynchCritSection* routine prevents the driver's ISR from executing.

  You should not write a single, large, general-purpose *SynchCritSection* routine with a **switch** statement or many nested **if..then..else** statements to determine what operations it will carry out or what state information to update. On the other hand, you should avoid writing numerous *SynchCritSection* routines that program only a single device register.

Accessing Shared State Information

6/25/2019 • 2 minutes to read • Edit Online

Use the following general guidelines for designing and writing *SynchCritSection* routines that maintain state:

- To access data that an ISR also accesses, a driver routine must call a *SynchCritSection* routine. Non-critical section code can be interrupted. Remember that it is not sufficient to simply acquire a spin lock to protect data that ISRs also access, because ISRs execute at DIRQL and acquiring a spin lock (**KeAcquireSpinLock**) only raises IRQL to DISPATCH_LEVEL, which allows an interrupt to invoke the ISR on the current processor.
- Give each *SynchCritSection* routine that maintains state information responsibility for a discrete set of state variables. That is, avoid writing *SynchCritSection* routines that maintain overlapping state information.

  This prevents contention, and possibly race conditions, between *SynchCritSection* routines (and the ISR) trying to access the same state concurrently.

  This also ensures that each *SynchCritSection* routine returns control as quickly as possible because one *SynchCritSection* routine never has to wait for another that updates some of the same state information to return control.
- Avoid writing a single, large, general-purpose *SynchCritSection* routine that does more testing of conditions to determine what to do than actually doing useful work. On the other hand, avoid having many *SynchCritSection* routines that never execute a conditional statement because each updates only a single byte of state information.
- Every *SynchCritSection* routine must return control as quickly as possible, because running any *SynchCritSection* routine prevents the driver's ISR from executing.

Following is a technique for maintaining a timer counter in a device extension. Assume the driver uses the counter to determine if an I/O operation has timed out. Also assume the driver does not overlap I/O operations.

- The driver's *StartIo* routine initializes the timer counter to some initial value for each I/O request. The driver then adds a second to its device time-out value, in case its *IoTimer* routine has just returned control.
- The driver's ISR must set this timer counter to minus one.
- The driver's *IoTimer* routine is called once per second to read the time counter and determine whether the ISR has already set it to minus one. If not, the *IoTimer* routine decrements the counter by using **KeSynchronizeExecution** to call a SynchCritSection_1 routine.

  If the counter goes to zero, indicating that the request timed out, the SynchCritSection_1 routine calls a SynchCritSection_2 routine to program a device reset operation. If the counter is minus one, the *IoTimer* routine simply returns.
- If the driver's *DpcForIsr* routine must reprogram the device to begin a partial-transfer operation, it must reinitialize the timer counter as the *StartIo* routine did.

  The *DpcForIsr* routine also must use **KeSynchronizeExecution** to call the SynchCritSection_2 routine, or possibly a SynchCritSection_3 routine, to program the device for another transfer operation.

In this scenario, the driver has more than one *SynchCritSection* routine, each with discrete, specific responsibilities; one to maintain its timer counter, and one or more others to program the device. Each *SynchCritSection* routine can return control quickly because it performs a single, discrete task.

Note that the driver has a single SynchCritSection_1 routine which, along with the driver's ISR, maintains the state

to the timer counter. Thus, there is no contention for access to the timer counter among several *SynchCritSection* routines and the ISR.

Managing Hardware Priorities

7/9/2019 • 4 minutes to read • Edit Online

The IRQL at which a driver routine executes determines which kernel-mode driver support routines it can call. For example, some driver support routines require that the caller be running at IRQL = DISPATCH_LEVEL. Others cannot be called safely if the caller is running at any IRQL higher than PASSIVE_LEVEL.

Following is a list of IRQLs at which the most commonly implemented standard driver routines are called. The IRQLs are listed from lowest to highest priority.

PASSIVE_LEVEL
Interrupts Masked Off — None.

Driver Routines Called at PASSIVE_LEVEL — **DriverEntry**, *AddDevice*, *Reinitialize*, *Unload* routines, most dispatch routines, driver-created threads, worker-thread callbacks.

APC_LEVEL
Interrupts Masked Off — APC_LEVEL interrupts are masked off.

Driver Routines Called at APC_LEVEL — Some dispatch routines (see Dispatch Routines and IRQLs).

DISPATCH_LEVEL
Interrupts Masked Off — DISPATCH_LEVEL and APC_LEVEL interrupts are masked off. Device, clock, and power failure interrupts can occur.

Driver Routines Called at DISPATCH_LEVEL — *StartIo*, *AdapterControl*, *AdapterListControl*, *ControllerControl*, *IoTimer*, *Cancel* (while holding the cancel spin lock), *DpcForIsr*, *CustomTimerDpc*, *CustomDpc* routines.

DIRQL
Interrupts Masked Off — All interrupts at IRQL <= DIRQL of driver's interrupt object. Device interrupts with a higher DIRQL value can occur, along with clock and power failure interrupts.

Driver Routines Called at DIRQL — *InterruptService*, *SynchCritSection* routines.

The only difference between APC_LEVEL and PASSIVE_LEVEL is that a process executing at APC_LEVEL cannot get APC interrupts. But both IRQLs imply a thread context and both imply that the code can be paged out.

Lowest-level drivers process IRPs while running at one of three IRQLs:

- PASSIVE_LEVEL, with no interrupts masked off on the processor, in the driver's Dispatch routine(s)

  DriverEntry, *AddDevice*, *Reinitialize*, and *Unload* routines also are run at PASSIVE_LEVEL, as are any driver-created system threads.

- DISPATCH_LEVEL, with DISPATCH_LEVEL and APC_LEVEL interrupts masked off on the processor, in the *StartIo* routine

  AdapterControl, *AdapterListControl*, *ControllerControl*, *IoTimer*, *Cancel* (while it holds the cancel spin lock), and *CustomTimerDpc* routines also are run at DISPATCH_LEVEL, as are *DpcForIsr* and *CustomDpc* routines.

- Device IRQL (DIRQL), with all interrupts at less than or equal to the *SynchronizeIrql* of the driver's interrupt object(s) masked off on the processor, in the ISR and *SynchCritSection* routines

Most higher-level drivers process IRPs while running at either of two IRQLs:

- PASSIVE_LEVEL, with no interrupts masked off on the processor, in the driver's dispatch routines

  DriverEntry, *Reinitialize*, *AddDevice*, and *Unload* routines also are run at PASSIVE_LEVEL, as are any driver-created system threads or worker-thread callback routines or file system drivers.

- DISPATCH_LEVEL, with DISPATCH_LEVEL and APC_LEVEL interrupts masked off on the processor, in the driver's *IoCompletion* routine(s)

  IoTimer, *Cancel*, and *CustomTimerDpc* routines also are run at DISPATCH_LEVEL.

In some circumstances, intermediate and lowest-level drivers of mass-storage devices are called at IRQL APC_LEVEL. In particular, this can occur at a page fault for which a file system driver sends an **IRP_MJ_READ** request to lower drivers.

Most standard driver routines are run at an IRQL that allows them simply to call the appropriate support routines. For example, a device driver must call **AllocateAdapterChannel** while running at IRQL DISPATCH_LEVEL. Since most device drivers call these routines from a *StartIo* routine, usually they are running at DISPATCH_LEVEL already.

Note that a device driver that has no *StartIo* routine because it sets up and manages its own queues of IRPs is not necessarily running at DISPATCH_LEVEL IRQL when it should call **AllocateAdapterChannel**. Such a driver must nest its call to **AllocateAdapterChannel** between calls to **KeRaiseIrql** and **KeLowerIrql** so that it runs at the required IRQL when it calls **AllocateAdapterChannel** and restores the original IRQL when the calling routine regains control.

When calling driver support routines, be aware of the following.

- Calling **KeRaiseIrql** with an input *NewIrql* value that is less than the current IRQL causes a fatal error. Calling **KeLowerIrql** except to restore the original IRQL (that is, after a call to **KeRaiseIrql**) also causes a fatal error.

- While running at IRQL >= DISPATCH_LEVEL, calling **KeWaitForSingleObject** or **KeWaitForMultipleObjects** for kernel-defined dispatcher objects to wait for a nonzero interval causes a fatal error.

- The only driver routines that can safely wait for events, semaphores, mutexes, or timers to be set to the signaled state are those that run in a nonarbitrary thread context at IRQL PASSIVE_LEVEL, such as driver-created threads, the **DriverEntry** and *Reinitialize* routines, or dispatch routines for inherently synchronous I/O operations (such as most device I/O control requests).

- Even while running at IRQL PASSIVE_LEVEL, pageable driver code must not call **KeSetEvent**, **KeReleaseSemaphore**, or **KeReleaseMutex** with the input *Wait* parameter set to **TRUE**. Such a call can cause a fatal page fault.

- Any routine that is running at greater than IRQL APC_LEVEL can neither allocate memory from paged pool nor access memory in paged pool safely. If a routine running at IRQL greater than APC_LEVEL causes a page fault, it is a fatal error.

- A driver must be running at IRQL DISPATCH_LEVEL when it calls **KeAcquireSpinLockAtDpcLevel** and **KeReleaseSpinLockFromDpcLevel**.

  A driver can be running at IRQL <= DISPATCH_LEVEL when it calls **KeAcquireSpinLock** but it must release that spin lock by calling **KeReleaseSpinLock**. In other words, it is a programming error to release a spin lock acquired with **KeAcquireSpinLock** by calling **KeReleaseSpinLockFromDpcLevel**.

  A driver must not call **KeAcquireSpinLockAtDpcLevel**, **KeReleaseSpinLockFromDpcLevel**, **KeAcquireSpinLock**, or **KeReleaseSpinLock** while running at IRQL > DISPATCH_LEVEL.

- Calling a support routine that uses a spin lock, such as an **ExInterlocked*Xxx*** routine, raises IRQL on the current processor either to DISPATCH_LEVEL or to DIRQL if the caller is not already running at a raised IRQL.
- Driver code that runs at IRQL > PASSIVE_LEVEL should execute as quickly as possible. The higher the IRQL at which a routine runs, the more important it is for good overall performance to tune that routine to execute as quickly as possible. For example, any driver that calls **KeRaiseIrql** should make the reciprocal call to **KeLowerIrql** as soon as it can.

For more information about determining priorities, see the Scheduling, Thread Context, and IRQL white paper.

Introduction to Plug and Play

6/25/2019 • 2 minutes to read • Edit Online

Plug and Play (PnP) is a combination of hardware and software support that enables a computer system to recognize and adapt to hardware configuration changes with little or no intervention by a user. A user can add devices to, and remove devices from, a computer system without having to do awkward and confusing manual configuration, and without having knowledge of intricate computer hardware. For example, a user can dock a portable computer and use the docking station keyboard, mouse, and monitor without making manual configuration changes.

PnP requires support from device hardware, system software, and drivers. Initiatives in the hardware industry define standards (such as the PnP ISA definition and the PC Card standard) for easy identification of add-in boards and basic system components. This Windows Driver Kit (WDK) documentation focuses on the system software support for PnP and how drivers use that support to implement PnP.

The system software support for PnP, together with PnP drivers provides the following:

- Automatic and dynamic recognition of installed hardware

  The system software recognizes hardware during initial system installation, recognizes PnP hardware changes that occur between system boots, and responds to run-time hardware events such as docking or undocking and device insertion or removal.

- Hardware resource allocation (and reallocation)

  The PnP manager determines the hardware resources requested by each device (for example, input/output ports [I/O], interrupt requests [IRQs], direct memory access [DMA] channels, and memory locations) and assigns hardware resources appropriately. The PnP manager reconfigures resource assignments when necessary, such as when a new device is added to the system that requires resources already in use.

  Drivers for PnP devices do not assign resources; instead, the requested resources for a device are identified when the device is enumerated. The PnP manager retrieves the requirements for each device during resource allocation. Resources are not dynamically configurable for legacy devices, so the PnP manager assigns resources to legacy devices first.

- Loading of appropriate drivers

  The PnP manager determines which drivers are required to support each device and loads those drivers.

- A programming interface for drivers to interact with the PnP system

  The interface includes I/O manager routines, Plug and Play minor IRPs, required standard driver routines, and information in the registry.

- Mechanisms for drivers and applications to learn of changes in the hardware environment and take appropriate actions

  PnP enables drivers and user-mode code to register for, and be notified of, certain hardware events.

PnP drivers are an important part of PnP support. For a driver to qualify as PnP it must provide the required PnP entry points, handle the required PnP IRPs, and follow PnP guidelines.

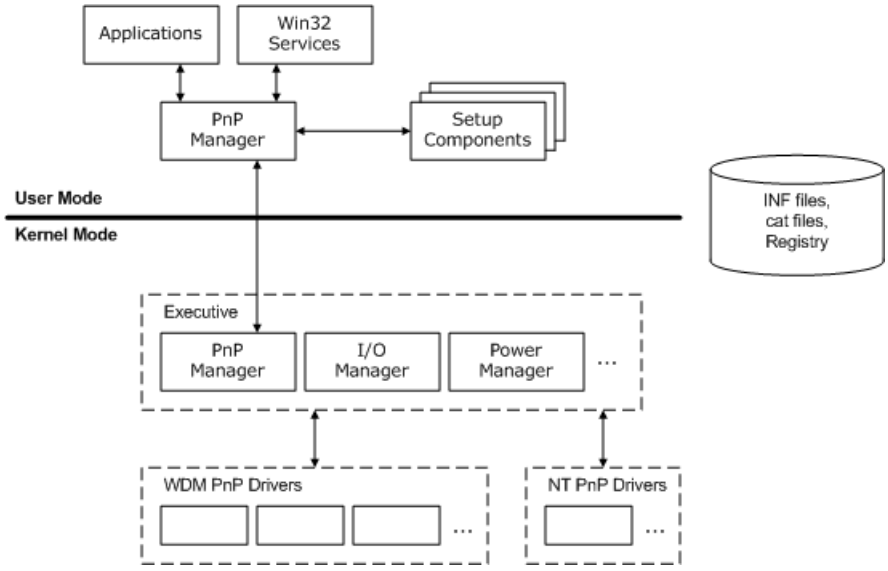
This section contains the following additional topics:

PnP Components

PnP Components

6/25/2019 • 2 minutes to read • Edit Online

The following figure shows the components that work together to support PnP.



The PnP manager has two parts: the kernel-mode PnP manager and the user-mode PnP manager. The kernel-mode PnP manager interacts with operating system components and drivers to configure, manage, and maintain devices. The user-mode PnP manager interacts with user-mode setup components, such as Class Installers, to configure and install devices. The user-mode PnP manager also interacts with applications to, for example, register an application for notification of device changes and notify the application when a device event occurs.

PnP drivers support the physical, logical, and virtual devices on a machine. The term "PnP driver" refers to any Windows driver that supports the interfaces described in this section. While most PnP drivers are also WDM drivers and thus source-compatible across Windows platforms, a few drivers support PnP without fully implementing WDM.

All drivers should support PnP and power management. If a single driver does not support PnP and power management, it constrains the PnP and power management support of the system as a whole.

See Device Installation Overview for information about device and driver setup, including (INF) files, CAT files, and the registry.

Levels of Support for PnP

12/5/2018 • 2 minutes to read • Edit Online

The extent to which a device supports PnP depends on the PnP support in both the device hardware and the device drivers (see the following table).

	PNP DRIVER	NON-PNP DRIVER
PnP Device	Full PnP	No PnP
Non-PnP Device	Possible partial PnP	No PnP

Any device that supports PnP should have PnP support in its drivers.

A non-PnP device can have some PnP capability if it is driven by a PnP driver. For example, an ISA sound card or an EISA network card can be manually installed and then a PnP driver can treat the card like a PnP device.

If a driver does not support PnP, its devices behave as non-PnP devices regardless of any hardware PnP support. A non-PnP driver can constrain the PnP and power management capabilities of the whole system.

Legacy drivers (that is, drivers written before the operating system supported PnP) continue to work as they did previously, without any PnP capability. New drivers should include PnP support.

PnP Driver Design Guidelines

6/25/2019 • 2 minutes to read • Edit Online

Plug and Play provides:

- Automatic and dynamic recognition of installed hardware
- Hardware resource allocation (and reallocation)
- Loading of appropriate drivers
- An interface for drivers to interact with the PnP system
- Mechanisms for drivers and applications to learn of changes in the hardware environment

To support PnP, a driver must follow these guidelines:

- It must contain a *DispatchPnP* routine.

  This dispatch routine must handle **IRP_MJ_PNP** requests and associated minor function codes. For more information, see DispatchPnP Routines.

- It must not search for hardware.

  The PnP manager is responsible for determining the presence of hardware devices. When the PnP manager detects a device, it notifies the driver by calling its *AddDevice* routine. Hardware can be detected when the system is booted, or any time that a user adds a device to, or removes one from, a running system.

- It must not allocate hardware resources.

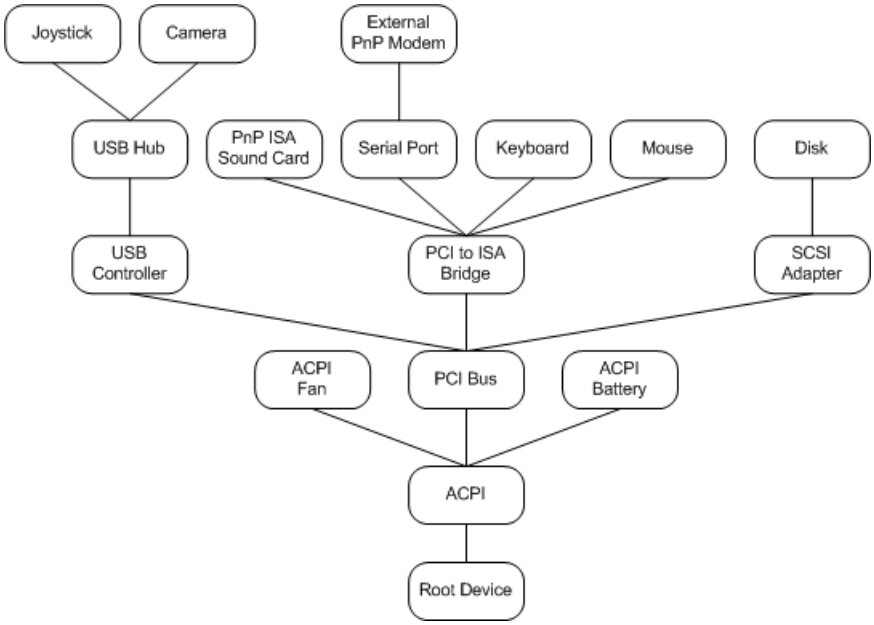
  A PnP driver must provide the PnP manager with lists of resources that a device can potentially use. The PnP manager is responsible for assigning resources to each device, and notifying the driver of each device's assignments when it sends an **IRP_MN_START_DEVICE** request. The driver must thus be capable of working with various configurations of hardware resources.

Some drivers are insulated from the details of the PnP and power management by system-supplied port or class drivers. For example, a SCSI port driver insulates a SCSI miniport driver from many of the details of the power and PnP systems, so a SCSI miniport driver does not need to handle power and PnP IRPs directly. For such drivers, see the driver-specific documentation for details of the required PnP support.

Device Tree

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager maintains a device tree that keeps track of the devices in the system. The following figure shows the device tree for a sample system configuration.



The device tree contains information about the devices present on the system. The PnP manager builds this tree when the machine boots, using information from drivers and other components, and updates the tree as devices are added or removed.

Each node of the device tree is called a device node, or *devnode*. A devnode consists of the *device objects* for the device's drivers, plus internal information maintained by the system. Therefore, there is a devnode for each *device stack*.

The PnP manager asks a bus driver for a list of its child devices using an **IRP_MN_QUERY_DEVICE_RELATIONS** request. The bus driver determines its list of children according to its bus protocol. For example, the Windows ACPI driver, Acpi.sys, looks in the ACPI namespace, the PCI driver queries PCI configuration space, and a USB hub driver follows the USB bus protocol.

The device tree is hierarchical, with devices on a bus represented as "children" of the bus adapter, controller or other *bus device*. (A bus device is any device to which other physical, logical, or virtual devices can be attached.) You can see the hierarchy of devices in the device tree using Device Manager and choosing the view option that allows you to view devices by connection.

The hierarchy of the device tree reflects the structure in which the devices are attached in the machine. The PnP manager uses this hierarchy as it manages the devices. For example, if a user requests to unplug the USB controller from the machine represented by the previous figure, the PnP manager determines from the device tree that this action would result in three other devices also being unplugged (the USB hub, the joystick, and the camera). When the PnP manager queries the drivers for the USB controller to determine if it is safe to remove the controller, it also queries the drivers of the controller's descendants (the hub, joystick, and camera).

The device tree is dynamic. As devices are added to, and removed from the machine, the PnP manager (together with drivers) maintains a current picture of the devices on the system.

There are other relationships between devices on the machine besides the hierarchical relationships represented

in the device tree. These include *removal relations* and *ejection relations*. See the reference page for **IRP_MN_QUERY_DEVICE_RELATIONS** for more information.

You cannot make any assumptions about the order in which the device tree is built, except that a bus device is configured before any of its child devices. For example, you should not assume that one device on a bus is configured before another device on the bus.

Hardware Resources

6/25/2019 • 5 minutes to read • Edit Online

Hardware resources are the assignable, addressable bus paths that allow peripheral devices and system processors to communicate with each other. Hardware resources typically include I/O port addresses, interrupt vectors, and blocks of bus-relative memory addresses.

Before the system can communicate with a *device instance*, the PnP manager must assign hardware resources to the device instance based on knowledge of which resources are available and which ones the device instance is capable of using. Resources are assigned to each device node in the device tree (assuming that the represented device needs resources and those resources are available). The PnP manager keeps track of hardware resources using lists, which it associates with device nodes. It uses two types of lists:

Resource Requirements List
Devices are typically designed to operate within ranges of resource assignments. For instance, a device might require only one interrupt vector, but it might be able to use any one of a range of vectors. For each device instance, the PnP manager maintains a *resource requirements list* that specifies all of the ranges of hardware resources in which the device can operate. The list's name stems from the fact that the PnP manager is required to choose resources from this list when assigning them to the device.

Kernel-mode code specifies resource requirements lists using **IO_RESOURCE_REQUIREMENTS_LIST** structures (either as input to system routines or in response to IRPs). User-mode code specifies resource requirements lists using PnP configuration manager structures as input to PnP configuration manager functions.

Resource List
When the PnP manager assigns resources to a device, it keeps track of these assignments by creating a list of assigned resources for each device instance. These lists could be called *resource assignment lists*, but that name is typically shorted to *resource lists*. The PnP manager can change resource list contents as devices are added to or removed from a system and resources are subsequently reallocated. (Resources can also be assigned by a PnP BIOS. Also, installation software—using INF files or user input—can force the PnP manager to assign specific resources to a device.)

Kernel-mode code specifies resource lists by using **CM_RESOURCE_LIST** structures (either as input to system routines or in response to IRPs). User-mode code specifies resource lists using PnP configuration manager structures as input to PnP configuration manager functions.

The PnP manager stores resource requirements lists and resource lists in the registry, where they can be viewed by using Regedit.exe. Drivers can access these lists indirectly through Plug and Play routines and Plug and Play Minor IRPs. User-mode applications can use PnP configuration manager functions. (Drivers and applications must not directly access these lists using registry functions because the storage format is subject to change in a future release.)

Logical Configurations

Both resource requirements lists and resource lists contain one or more *logical configurations*. Each logical configuration identifies either a range of acceptable resources, or a set of specific resources for a specific *device instance*. Additionally, each logical configuration for a device instance belongs to one of the *logical configuration types*. Configuration types are listed below. Several logical configurations, of the same or different types, might be assigned to each device instance.

Logical Configuration Types for Resource Requirements Lists

Basic Configuration
A resource requirements list identifying resource ranges supplied by a Plug and Play device. A driver should

return this list when it receives the **IRP_MN_QUERY_RESOURCE_REQUIREMENTS** IRP. (The basic configuration for a non-PnP device can be described in an INF file. In this case, device installation software reads the INF file and calls PnP configuration manager functions to create a requirements list.)

Filtered Configuration
A resource requirements list that has been supplied to a driver stack, possibly modified, then returned by the driver stack, in response to the **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** IRP. The PnP manager uses the resulting filtered configuration as the basis for allocating resources.

Override Configuration
A resource requirements list that overrides basic configurations. Typically, a device installer creates an override configuration if the device's INF file includes an **INF DDInstall.LogConfigOverride section**. An override configuration is not removed if its device is physically removed from the system.

Logical Configuration Types for Resource Lists

Boot Configuration
A resource list identifying the resources assigned to a device instance when the system is booted. (For PnP devices, this is the configuration supplied by the BIOS; for non-PnP devices, these resources might be selected by jumpers on the card.) A driver should return this resource list when it receives the **IRP_MN_QUERY_RESOURCES** IRP. (A boot configuration can be partially empty if the BIOS cannot determine all resources used by a device.) The PnP manager can modify this list if a device is removed or restarted. For non-PnP devices, this configuration type can be used instead of a forced configuration, in which case it has a lower configuration priority than an equivalent forced configuration. Only one boot configuration can exist for each device instance.

Forced Configuration
A resource list identifying resources that a device instance must use. A forced configuration prevents the PnP manager from assigning other resources to the device instance. A device installer might create a forced configuration based on information that is either contained in an INF or received from a user. A forced configuration is not removed if its device is physically removed from the system. Only one forced configuration can exist for each device instance.

Allocated Configuration
A resource list identifying resources currently in use by a device instance. Only one allocated configuration can exist for each device instance.

Device drivers are responsible for determining a PnP-compatible device's basic configuration, filtered configuration, and boot configuration, and for returning that information in response to IRPs sent by the PnP manager. (For more information, see Adding a PnP Device to a Running System.) Driver installation software can create override configurations, forced configurations, and, for non-PnP devices, boot configurations. The PnP manager maintains each device instance's allocated configuration.

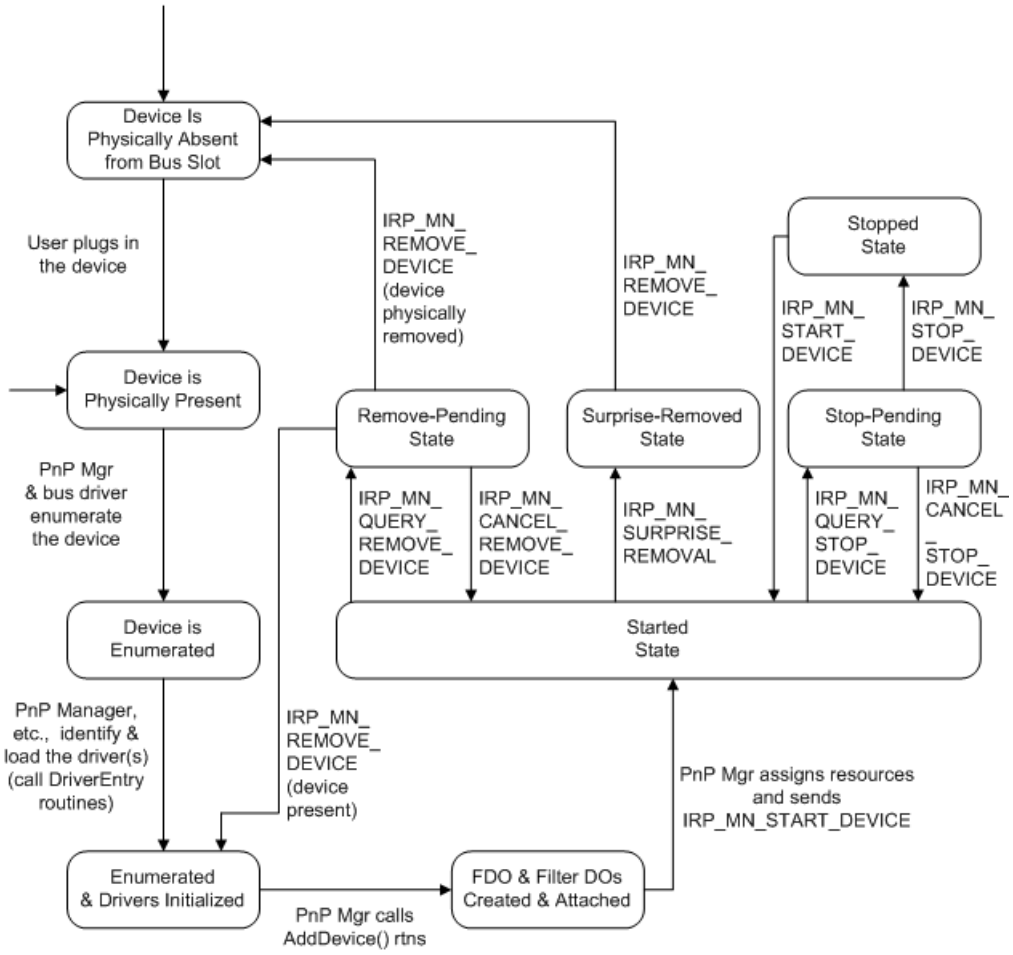
A priority is assigned to each configuration when it is created. If the PnP manager finds that a device instance has been assigned several logical configurations of the same type, it attempts to use the one with the highest priority first. If that configuration results in resource conflicts, it tries the configuration with the next lower priority. (For a list of configuration priorities, see **CM_Add_Empty_Log_Conf**.)

State Transitions for PnP Devices

6/25/2019 • 2 minutes to read • Edit Online

On a PnP system, a device transitions through various PnP states as it is configured, started, possibly stopped to rebalance resources, and possibly removed. This section provides an overview of the PnP device states. The overview is a road map for much of the PnP support required in a driver. Other parts of this documentation describe each state transition in detail.

The following figure shows the PnP states for a device and how a device transitions from one state to another.



Starting at the top left of the previous figure, a PnP device is physically present in the system because either the user just inserted the device or the device was present at boot time. The device is not yet known to the system software.

To begin software configuration for the device, the PnP manager and the parent bus driver enumerate the device. The PnP manager, possibly with help from user-mode components, identifies the drivers for the device, including the function driver and any optional filter drivers. The PnP manager calls the **DriverEntry** routine of each driver if the driver is not yet loaded. For more information about reporting and enumerating a PnP device, see Adding a PnP Device to a Running System.

Once a driver is initialized, it must be ready to initialize its devices. The PnP manager calls a driver's *AddDevice* routine for each device the driver controls.

When a driver receives an **IRP_MN_START_DEVICE** request from the PnP manager, the driver starts the device and is ready to process I/O requests for the device. For information about handling an **IRP_MN_START_DEVICE** request, see Starting a Device.

If the PnP manager must reconfigure the hardware resources of an active device, it sends **IRP_MN_QUERY_STOP_DEVICE** and **IRP_MN_STOP_DEVICE** requests to the device's drivers. After it reconfigures the hardware resources, the PnP manager directs the drivers to restart the device by sending an **IRP_MN_START_DEVICE** request. For information about handling stop IRPs, see Stopping a Device. (The drivers for a boot-configured device can receive **IRP_MN_QUERY_STOP_DEVICE** and **IRP_MN_STOP_DEVICE** requests before the device has been started, although this step is not shown in the previous figure.)

On Windows 98/Me, the PnP manager also sends **IRP_MN_QUERY_STOP_DEVICE** and **IRP_MN_STOP_DEVICE** requests when a device is being disabled. Drivers on these systems also receive an **IRP_MN_STOP_DEVICE** request after a failed start.

When a PnP device is being physically removed from the system or has already been removed, the PnP manager sends various remove IRPs to the device's drivers, directing them to remove the device's software representation (device objects, and so forth). For information about handling remove IRPs, see Removing a Device.

At some point after all of a driver's devices have been removed, the PnP manager calls the driver's *Unload* routine and unloads the driver.

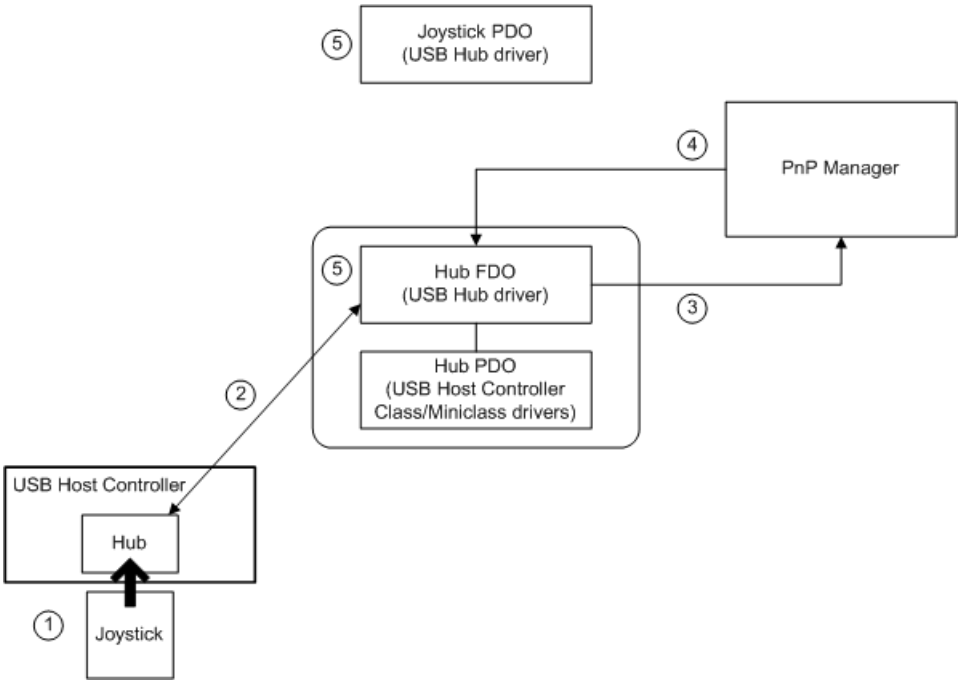
Adding a PnP Device to a Running System

6/25/2019 • 11 minutes to read • Edit Online

This section describes the sequence of events that occur when the system configures a PnP device that a user has added to a running machine. This discussion highlights the roles of the PnP manager, bus drivers, and function and filter drivers in enumerating and configuring a new device.

Most of this discussion is also relevant to configuring a PnP device that is present when the machine is booted. Specifically, devices whose drivers are marked SERVICE_DEMAND_START in an INF file are configured in essentially the same way whether the device is added dynamically or is present at boot time.

The following figure shows the first steps in configuring the device, starting from when the user plugs the hardware into the machine.



The following notes correspond to the circled numbers in the previous figure:

1. A user plugs a PnP device into a free slot on a PnP bus.

   In this example, the user plugs a PnP USB joystick into the hub on a USB host controller. The USB hub is a PnP bus device because child devices can be attached to it.

2. The function driver for the bus device determines that a new device is on its bus.

   How the driver determines this depends on the bus architecture. For some buses, the bus function driver receives hot-plug notification of new devices. If the bus does not support hot-plug notification, the user must take appropriate action in Control Panel to cause the bus to be enumerated.

   In this example, the USB bus supports hot-plug notification so the function driver for the USB bus is notified that its children have changed.

3. The function driver for the bus device notifies the PnP manager that its set of child devices has changed.

   The function driver notifies the PnP manager by calling **IoInvalidateDeviceRelations** with a *Type* of **BusRelations**.

4. The PnP manager queries the bus's drivers for the current list of devices on the bus.

The PnP manager sends an **IRP_MN_QUERY_DEVICE_RELATIONS** request to the device stack for the bus. The **Parameters.QueryDeviceRelations.Type** value is **BusRelations**, indicating that the PnP manager is asking for the current list of devices present on the bus (*bus relations*).

The PnP manager sends the IRP to the top driver in the device stack for the bus. According to the rules for PnP IRPs, each driver in the stack handles the IRP, if appropriate, and passes the IRP down to the next driver.

5. The function driver for the bus device handles the IRP.

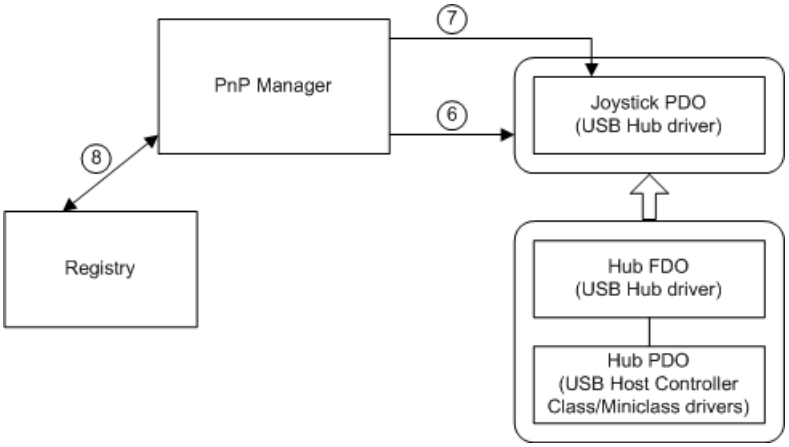
   See the reference page for **IRP_MN_QUERY_DEVICE_RELATIONS** for detailed information about handling this IRP.

   In this example, the USB hub driver handles this IRP for the hub *FDO*. The hub driver creates a *PDO* for the joystick device and includes a referenced pointer to the joystick PDO in its list of child devices returned with the IRP.

   When the USB hub's parent bus driver (the USB host controller class/miniclass driver pair) completes the IRP, the IRP travels back up the device stack by means of any *IoCompletion* routines registered by the hub drivers.

Note that the bus function driver reports a change in its list of children by requesting that the PnP manager query for its list of child devices. The resulting **IRP_MN_QUERY_DEVICE_RELATIONS** request is seen by all the drivers for the bus device. Typically, the bus function driver is the only driver to handle the IRP and report children. In some device stacks, a bus filter driver is present and participates in constructing the list of bus relations. One example is ACPI, which attaches as a bus filter driver for ACPI devices. In some device stacks, nonbus filter drivers handle the **IRP_MN_QUERY_DEVICE_RELATIONS** request, but this is not typical.

At this point, the PnP manager has the current list of devices on the bus. The PnP manager then determines whether any devices are newly arrived or have been removed. In this example, there is one new device. The following figure shows the PnP manager creating a devnode for the new device and beginning to configure the device.



The following notes correspond to the circled numbers in the previous figure:

1. The PnP manager creates devnodes for any new child devices on the bus.

   The PnP manager compares the list of bus relations returned in the **IRP_MN_QUERY_DEVICE_RELATIONS** IRP to the list of children for the bus currently recorded in the PnP device tree. The PnP manager creates a devnode for each new device and initiates removal processing for any devices that have been removed.

   In this example, there is one new device (a joystick), so the PnP manager creates a devnode for the joystick. At this point, the only driver that is configured for the joystick is the parent USB hub bus driver, which

created the joystick's PDO. Any optional bus filter drivers would also be present in the device stack, but the example omits bus filter drivers for simplicity.

The wide arrow between the two devnodes in the previous figure indicates that the joystick devnode is a child of the USB hub devnode.

2. The PnP manager gathers information about the new device and begins configuring the device.

   The PnP manager sends a sequence of IRPs to the device stack to gather information about the device. At this point, the device stack consists of only the PDO created by the device's parent bus driver and filter DOs for any optional bus filter drivers. Therefore, the bus driver and bus filter drivers are the only drivers that respond to these IRPs. In this example, the only driver in the joystick device stack is the parent bus driver, the USB hub driver.

   The PnP manager gathers information about a new device by sending IRPs to the device stack. These IRPs include the following:

   - **IRP_MN_QUERY_ID**, a separate IRP for each of the following types of hardware IDs:

     BusQueryDeviceID

     BusQueryInstanceID

     BusQueryHardwareIDs

     BusQueryCompatibleIDs

     BusQueryContainerID

   - **IRP_MN_QUERY_CAPABILITIES**
   - **IRP_MN_QUERY_DEVICE_TEXT**, a separate IRP for each of the following items:

     DeviceTextDescription

     DeviceTextLocationInformation

   - **IRP_MN_QUERY_BUS_INFORMATION**
   - **IRP_MN_QUERY_RESOURCES**
   - **IRP_MN_QUERY_RESOURCE_REQUIREMENTS**

   The PnP manager sends the IRPs listed above at this stage of processing a new PnP device, but not necessarily in the order listed, so you should not make assumptions about the order in which the IRPs are sent. Also, you should not assume that the PnP manager sends only the IRPs listed above.

   The PnP manager checks the registry to determine whether the device has been installed on this machine previously. The PnP manager checks for an <*enumerator*>\<*deviceID*> subkey for the device under the **Enum** branch. In this example, the device is new and must be configured "from scratch."

3. The PnP manager stores information about the device in the registry.

   The registry's **Enum** branch is reserved for use by operating system components and its layout is subject to change. Driver writers must use system routines to extract information related to drivers. Do not access the **Enum** branch directly from a driver. The following **Enum** information is listed for debugging purposes only.

   - The PnP manager creates a subkey for the device under the key for the device's enumerator.

     The PnP manager creates a subkey named **HKLM\System\CurrentControlSet\Enum**<*enumerator*>****<*deviceID*>. It creates the <*enumerator*> subkey if it does not already exist.

An *enumerator* is a component that discovers PnP devices based on a PnP hardware standard. The tasks of an enumerator are carried out by a PnP bus driver in partnership with the PnP manager. A device is typically enumerated by its parent bus driver, such as PCI or PCMCIA. Some devices are enumerated by a bus filter driver, such as ACPI.

- The PnP manager creates a subkey for this instance of the device.

  If **Capabilities.UniqueID** is returned as **TRUE** for **IRP_MN_QUERY_CAPABILITIES**, the device's unique ID is unique across the system. If not, the PnP manager modifies the ID so that it is unique system-wide.

  The PnP manager creates a subkey named **HKLM\System\CurrentControlSet\Enum**<*enumerator*>****<*deviceID*>****<*instanceID*>.

- The PnP manager writes information about the device to the subkey for the device instance.

  The PnP manager stores information, including the following, if it was supplied for the device:

  DeviceDesc — from **IRP_MN_QUERY_DEVICE_TEXT**

  Location — from **IRP_MN_QUERY_DEVICE_TEXT**

  Capabilities — the flags from **IRP_MN_QUERY_CAPABILITIES**

  UINumber — from **IRP_MN_QUERY_CAPABILITIES**

  HardwareID — from **IRP_MN_QUERY_ID**

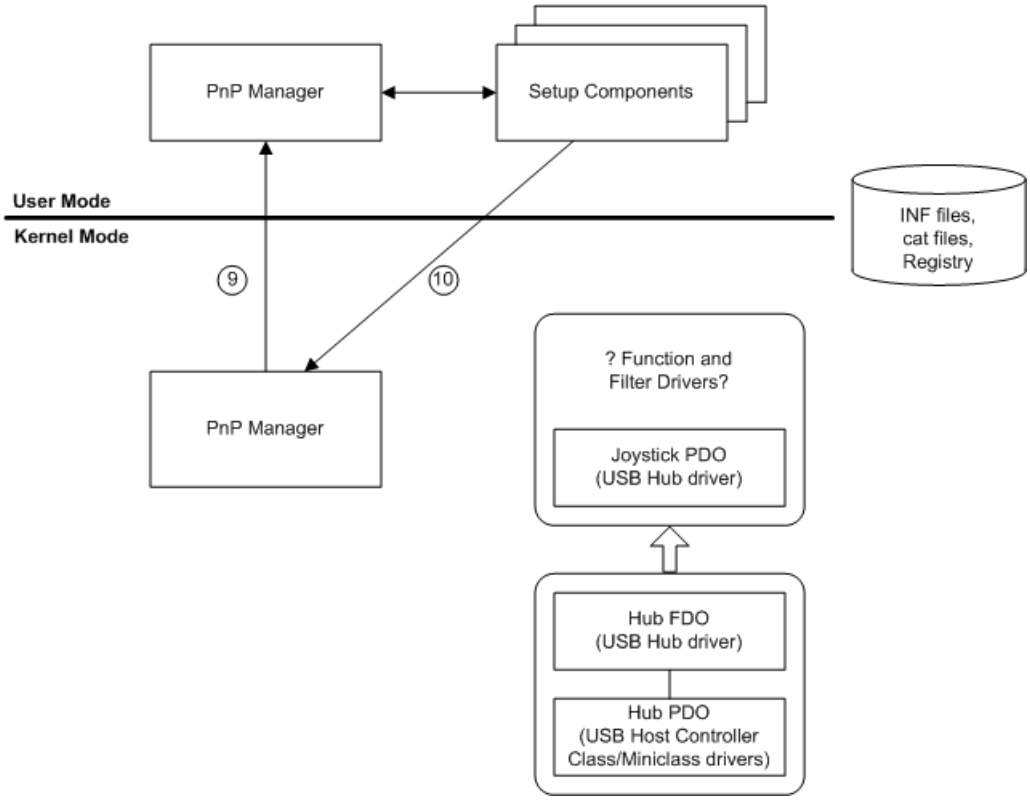
  CompatibleIDs — from **IRP_MN_QUERY_ID**

  ContainerID — from **IRP_MN_QUERY_ID**

  LogConf\BootConfig — from **IRP_MN_QUERY_RESOURCES**

  LogConf\BasicConfigVector — from **IRP_MN_QUERY_RESOURCE_REQUIREMENTS**

At this point, the PnP manager is ready to locate the function driver and filter drivers for the device, if any. (See the following figure.)



The following notes correspond to the numbered circles in the previous figure:

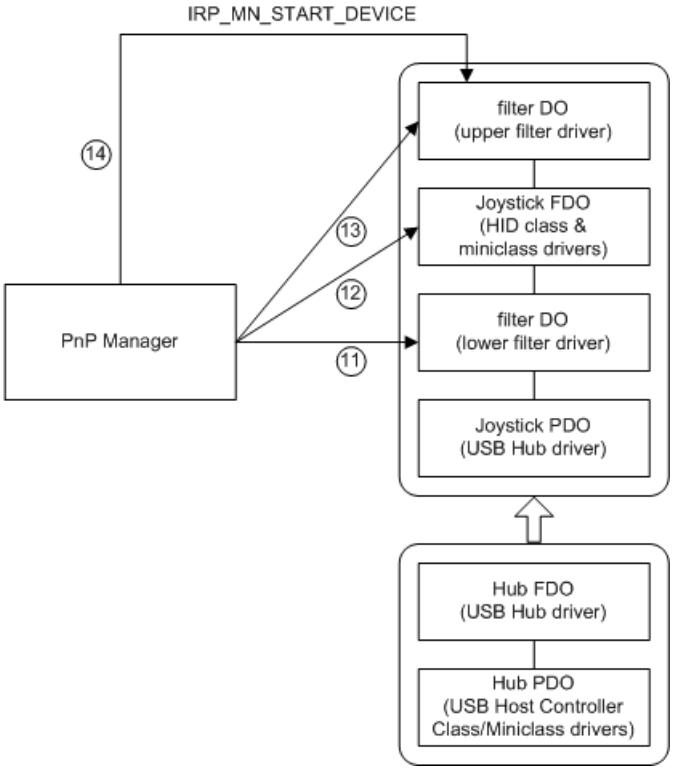
1. The kernel-mode PnP manager coordinates with the user-mode PnP manager and user-mode Setup components to find the function and filter drivers for the device, if there are any.

   The kernel-mode PnP manager queues an event to the user-mode PnP manager, identifying a device that needs to be installed. Once a privileged user logs in, the user-mode components proceed with finding drivers. See the device installation overview For information about Setup components and their role in installing a device.

2. The user-mode Setup components direct the kernel-mode PnP manager to load the function and filter drivers.

   The user-mode components call back to kernel mode to get the drivers loaded, causing their *AddDevice* routines to be called.

The following figure shows the PnP manager loading the drivers (if appropriate), calling their *AddDevice* routines, and directing the drivers to start the device.



The following notes correspond to the numbered circles in the previous figure:

1. Lower-filter drivers

   Before the function driver attaches to the device stack, the PnP manager processes any lower-filter drivers. For each lower-filter driver, the PnP manager calls the driver's **DriverEntry** routine if the driver is not yet loaded. Then the PnP manager calls the driver's *AddDevice* routine. In its *AddDevice* routine, the filter driver creates a filter device object (filter DO) and attaches it to the device stack (**IoAttachDeviceToDeviceStack**). Once it attaches its device object to the device stack, the driver is engaged as a driver for the device.

   In the USB joystick example, there is one lower-filter driver for the device.

2. Function driver

   After any lower filters are attached, the PnP manager processes the function driver. The PnP manager calls the function driver's **DriverEntry** routine if the driver is not yet loaded and calls the function driver's *AddDevice* routine. The function driver creates a function device object (FDO) and attaches it to the device

stack.

In this example, the function driver for the USB joystick is actually a pair of drivers: the HID class driver and the HID miniclass driver. The two drivers work together to serve as the function driver. The driver pair creates only one FDO and attaches it to the device stack.

3. Upper-filter drivers

   After the function driver is attached, the PnP manager processes any upper-filter drivers.

   In this example, there is one upper-filter driver for the device.

4. Assigning resources and starting the device

   The PnP manager assigns resources to the device, if needed, and issues an IRP to start the device.

   - Assigning resources

     Earlier in the configuration process, the PnP manager gathered the hardware resource requirements for the device from the device's parent bus driver. After the full set of drivers is loaded for the device, the PnP manager sends an **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** request to the device stack. All drivers in the stack have the opportunity to handle this IRP and modify the device's resource requirements list, if necessary.

     The PnP manager assigns resources to the device, if the device requires any, based on the device's requirements and the resources currently available.

     The PnP manager might need to rearrange the resource assignments of existing devices to satisfy the needs of the new device. This reassignment of resources is called "rebalancing." The drivers for the existing devices receive a sequence of stop and start IRPs during a rebalance, but the rebalance must be transparent to users.

     In the example of the USB joystick, USB devices do not require hardware resources so the PnP manager sets the resource list to **NULL**.

   - Starting the device (**IRP_MN_START_DEVICE**)

     Once the PnP manager assigns resources to the device, it sends an **IRP_MN_START_DEVICE** IRP to the device stack to direct the drivers to start the device.

   After the device is started, the PnP manager sends three more IRPs to the drivers for the device:

   - **IRP_MN_QUERY_CAPABILITIES**

     After the start IRP completes successfully, the PnP manager sends another **IRP_MN_QUERY_CAPABILITIES** IRP to the device stack. All the drivers for the device have the option of handling the IRP. The PnP manager sends this IRP at this time, after all drivers are attached and the device is started, because the function or filter drivers might need to access the device to collect capability information.

   - **IRP_MN_QUERY_PNP_DEVICE_STATE**

     This IRP gives a driver the opportunity to, for example, report that the device should not be displayed in user interfaces such as Device Manager and the Hotplug program. This is useful for devices that are present on a system but are not usable in the current configuration, such as a game port on a laptop that is not usable when the laptop is undocked.

   - **IRP_MN_QUERY_DEVICE_RELATIONS** for bus relations

     The PnP manager sends this IRP to determine whether the device has any child devices. If so, the PnP manager configures each child device.

Using GUID_PNP_LOCATION_INTERFACE

The GUID_PNP_LOCATION_INTERFACE interface supplies the SPDRP_LOCATION_PATHS Plug and Play (PnP) device property for a device.

To implement this interface in your driver, handle the IRP_MN_QUERY_INTERFACE IRP with InterfaceType = GUID_PNP_LOCATION_INTERFACE. Your driver supplies a pointer to a PNP_LOCATION_INTERFACE structure that contains pointers to the individual routines of the interface. The PnpGetLocationString routine provides the device-specific part of the device's SPDRP_LOCATION_PATHS property.

Plug and Play Minor IRPs

9/10/2019 • 2 minutes to read • Edit Online

This section describes the PnP IRPs that are sent to drivers. All PnP IRPs have the major function code **IRP_MJ_PNP** and a minor function code indicating the particular PnP request.

This section provides reference information for the individual IRPs. See Plug and Play for a description of the order in which the IRPs are sent, a discussion of how to handle IRPs in DispatchPnP routines, and a general discussion of PnP concepts and terminology.

For each IRP and each kind of driver, a driver is either required to handle the IRP, can optionally handle the IRP, or must not handle the IRP. Consult the table below to identify which IRPs your driver will handle and then consult the reference pages for information about the individual IRPs. The IRPs are listed in functional order in the table and in alphabetical order in the IRP reference pages.

If an IRP is marked "No" in the table for a particular driver, that driver must not handle the IRP. The driver must pass the IRP to the next driver in the device stack as described in the reference page for the IRP.

The PnP manager sends these IRPs. PnP drivers can send some of these IRPs, but only those so noted in this section.

The following are the minor function codes for PnP IRPs, and the driver types that handle them:

PNP IRP MINOR FUNCTION CODE	FUNCTION OR FILTER DRIVER FOR NONBUS DEVICE	FUNCTION DRIVER FOR BUS DEVICE (FOR BUS FDO)	BUS DRIVER OR BUS FILTER DRIVER (FOR CHILD PDOS)	
IRP_MN_START_DEVICE	Required	Required	Required	
IRP_MN_QUERY_STOP_DEVICE	Required	Required	Required	
IRP_MN_STOP_DEVICE	Required	Required	Required	
IRP_MN_CANCEL_STOP_DEVICE	Required	Required	Required	
IRP_MN_QUERY_REMOVE_DEVICE	Required	Required	Required	
IRP_MN_REMOVE_DEVICE	Required	Required	Required	
IRP_MN_CANCEL_REMOVE_DEVICE	Required	Required	Required	
IRP_MN_SURPRISE_REMOVAL	Required	Required	Required	
IRP_MN_QUERY_CAPABILITIES	Optional	Optional	Required	

PNP IRP MINOR FUNCTION CODE	FUNCTION OR FILTER DRIVER FOR NONBUS DEVICE	FUNCTION DRIVER FOR BUS DEVICE (FOR BUS FDO)	BUS DRIVER OR BUS FILTER DRIVER (FOR CHILD PDOS)	
IRP_MN_QUERY_PNP_DEVICE_STATE	Optional	Optional	Optional	
IRP_MN_FILTER_RESOURCE_REQUIREMENTS	Optional (1)	Optional (1)	No	
IRP_MN_DEVICE_USAGE_NOTIFICATION	Required (1)	Required (1)	Required (1)	
IRP_MN_QUERY_DEVICE_RELATIONS				
- **BusRelations**	Optional (1)	Required	No (2)	
- **EjectionRelations**	No	No	Optional	
- **RemovalRelations**	Optional	Optional	No	
- **TargetDeviceRelation**	No	No	Required	
IRP_MN_QUERY_RESOURCES	No	No	Required (1)	
IRP_MN_QUERY_RESOURCE_REQUIREMENTS	No	No	Required (1)	
IRP_MN_QUERY_ID				
- **BusQueryDeviceID**	No	No	Required	
- **BusQueryHardwareIDs**	No	No	Optional	
- **BusQueryCompatibleIDs**	No	No	Optional	
- **BusQueryInstanceID**	No	No	Optional	
- **BusQueryContainerID**	No	No	Required (3)	
IRP_MN_QUERY_DEVICE_TEXT	No	No	Required (1)	

PNP IRP MINOR FUNCTION CODE	FUNCTION OR FILTER DRIVER FOR NONBUS DEVICE	FUNCTION DRIVER FOR BUS DEVICE (FOR BUS FDO)	BUS DRIVER OR BUS FILTER DRIVER (FOR CHILD PDOS)	
IRP_MN_QUERY_BUS_INFORMATION	No	No	Required (1)	
IRP_MN_QUERY_INTERFACE	Optional	Optional	Required (1)	
IRP_MN_READ_CONFIG	No	No	Required (1)	
IRP_MN_WRITE_CONFIG	No	No	Required (1)	
IRP_MN_DEVICE_ENUMERATED	No	No	Required (1)	
IRP_MN_SET_LOCK	No	No	Required (1)	

(1) Required or optional in certain situations. See the reference page for the IRP for more details.

(2) Bus filter drivers might handle a query for **BusRelations**.

(3) Supported in Windows 7 and later versions of Windows.

IRP_MN_CANCEL_REMOVE_DEVICE

6/25/2019 • 2 minutes to read • Edit Online

All PnP drivers must handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP to inform the drivers for a device that the device will not be removed.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input Parameters

None

Output Parameters

None

I/O Status Block

A driver must set **Irp->IoStatus.Status** to STATUS_SUCCESS for this IRP. If a driver fails this IRP, the device is left in an inconsistent state.

Operation

This IRP must be handled first by the parent bus driver for a device and then by each higher driver in the device stack.

In response to this IRP, drivers return the device to the state it was in prior to receiving the **IRP_MN_QUERY_REMOVE_DEVICE** request.

If the device is already started when the driver receives this IRP, the driver simply sets status to success and passes the IRP to the next driver (or completes the IRP if the driver is a bus driver). For such a cancel-remove IRP, a function or filter driver need not set a completion routine. The device may not be in the remove-pending state, because, for example, the driver failed the previous **IRP_MN_QUERY_REMOVE_DEVICE**.

The PnP manager calls any **EventCategoryTargetDeviceChange** notification callbacks with GUID_TARGET_DEVICE_REMOVE_CANCELLED after the **IRP_MN_CANCEL_REMOVE_DEVICE** request completes. Such callbacks were registered on the device by calling **IoRegisterPlugPlayNotification**. The PnP manager also calls any user-mode components that registered for notification on the device by calling **RegisterDeviceNotification**.

If a file system is mounted on the device, it must undo any operations it did in response to the query-remove notification.

See Plug and Play for detailed information about handling remove IRPs and for the general rules for handling all Plug and Play minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IoRegisterPlugPlayNotification

IRP_MN_QUERY_REMOVE_DEVICE

IRP_MN_CANCEL_STOP_DEVICE

6/25/2019 • 2 minutes to read • Edit Online

All PnP drivers must handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP, at some point after an **IRP_MN_QUERY_STOP_DEVICE**, to inform the drivers for a device that the device will not be disabled (Windows 98/Me only) or stopped for resource reconfiguration.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input Parameters

None

Output Parameters

None

I/O Status Block

A driver must set **Irp->IoStatus.Status** to STATUS_SUCCESS for this IRP. If a driver fails this IRP, the device is left in an inconsistent state.

Operation

This IRP must be handled first by the parent bus driver for a device and then by each higher driver in the device stack.

In response to this IRP, drivers return the device to the started state. Drivers start any IRPs that were held while the device was in the stop-pending state.

If the device is already in an active state when the driver receives this IRP, a function or filter driver simply sets status to success and passes the IRP to the next driver. The parent bus driver completes the IRP. For such a cancel-stop IRP, a function or filter driver need not set a completion routine.

See Plug and Play for detailed information about handling stop IRPs and for the general rules for handling all Plug and Play minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IRP_MN_QUERY_STOP_DEVICE

IRP_MN_DEVICE_ENUMERATED

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager uses this I/O request packet (IRP) to notify bus drivers that a device object exists and that it has been fully enumerated by the plug and play manager.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP just before user mode is notified with GUID_DEVICE_ENUMERATED. This IRP allows drivers to provide a preprocess routine for IRP_MN_DEVICE_ENUMERATED, such as filling in additional device properties. This IRP primarily allows drivers to set device properties for the physical device object (PDO) by using **IoSetDevicePropertyData**.

Input Parameters

None

Output Parameters

None

I/O Status Block

A driver that handles this IRP sets Irp->IoStatus.Status to STATUS_SUCCESS or an appropriate error status.

Operation

The **IRP_MN_DEVICE_ENUMERATED** IRP is sent to the bus driver's PDO to indicate that the bus driver PDO exists.

Sending the IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Version	Available in Windows 7 and later versions of Windows.
Header	Wdm.h

See also

Plug and Play Minor IRPs

IRP_MN_DEVICE_USAGE_NOTIFICATION

6/25/2019 • 6 minutes to read • Edit Online

System components send this IRP to ask the drivers for a device whether the device can support a *special file*. Special files include paging files, dump files, and hibernation files. If all the drivers for the device succeed the IRP, the system creates the special file. The system also sends this IRP to inform drivers that a special file has been removed from the device.

Function drivers must handle this IRP if their device can contain a paging file, dump file, or hibernation file. Filter drivers must handle this IRP if the function driver they are filtering handles the IRP. Bus drivers must handle this IRP for their adapter or controller (bus FDO) and for their child devices (child PDOs).

Major Code

IRP_MJ_PNP When Sent

The system sends this IRP when it is creating or deleting a paging file, dump file, or hibernation file. If a device has a power management relationship that falls outside of the conventional parent-child relationship, the driver can send this IRP to propagate device usage information to another device stack. For more information, see the description of the **PowerRelations** request in **IRP_MN_QUERY_DEVICE_RELATIONS**.

System components and drivers send this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

The **Parameters.UsageNotification.InPath** member of the **IO_STACK_LOCATION** structure is a BOOLEAN. When this parameter is **TRUE**, the system is creating a paging, crash dump, or hibernation file on the device. When **InPath** is **FALSE**, such a file has been removed from the device.

Parameters.UsageNotification.Type is an enum indicating the kind of file. This parameter has one of the following values: **DeviceUsageTypePaging**, **DeviceUsageTypeDumpFile**, or **DeviceUsageTypeHibernation**.

Output Parameters

None

I/O Status Block

Drivers set **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

Drivers do not modify the **Irp->IoStatus.Information** field; it remains at zero, as set by the component sending the IRP.

Operation

A driver handles this IRP on the IRP's way down the device stack and on the IRP's way back up the stack.

A driver responds to this IRP with a procedure like the following:

- If **Parameters.UsageNotification.InPath** is **TRUE**, determine whether the device supports the special file.

A driver should test for the specific **Parameters.UsageNotification.Type**(s) that the driver can support. Additional notification types might be added in the future.

See further information below describing the actions required to support each notification type.

If **Parameters.UsageNotification.InPath** is **TRUE** and the driver cannot support the special file on the device, the driver must complete the IRP with a failure status.

- If the device supports the special file:

  1. Take appropriate actions to reflect that the device now contains, or no longer contains, a special file.

     A driver typically increments or decrements a counter. For example, if **Parameters.UsageNotification.Type** is **DeviceUsageTypePaging** and **Parameters.UsageNotification.InPath** is **TRUE**, increment a count of the number of paging files on the device. Certain driver dispatch routines must check the counter(s).

     A device that contains a special file should not be disabled. A driver can call **IoInvalidateDeviceState**, requesting the PnP manager to re-query for the device's PnP device state information. In response to the resulting **IRP_MN_QUERY_PNP_DEVICE_STATE** IRP, the driver should set the PNP_DEVICE_NOT_DISABLEABLE flag.

     If **InPath** is **FALSE**, a driver sets the DO_POWER_PAGABLE bit in its device object for the device.

  2. Propagate the device usage information to any related devices that require the information.

     As part of its handling of an **IRP_MN_DEVICE_USAGE_NOTIFICATION** IRP, a driver might be required to pass the information to one or more other device stacks. Such a driver creates a new **IRP_MN_DEVICE_USAGE_NOTIFICATION** IRP (or IRPs) and sends them to the appropriate device stack (or stacks). The driver must wait for completion of any device-usage-notification IRP(s) that it sends before the driver finishes processing the device-usage IRP that it received.

     How to identify the related devices is device- and driver-specific. Typically, a driver sends the IRP to other drivers to which it would send I/O requests for the file. When a bus driver handles this request for a child device, it must send a usage notification IRP to the device stack for the device's parent.

     For example, when ftdisk is running a five-disk stripe set, it propagates paging notifications to each of these five disks, since each of these devices can be required to handle paging file operations.

  3. In a function or filter driver, set an *IoCompletion* routine.

  4. In a function or filter driver, set **Irp->IoStatus.Status** to STATUS_SUCCESS, set up the next stack location, and pass the IRP to the next lower driver with **IoCallDriver**. Do not complete the IRP.

     In a bus driver that is handling the IRP for a child PDO: set **Irp->IoStatus.Status** and complete the IRP (**IoCompleteRequest**).

  5. During IRP completion processing:

     If an *IoCompletion* routine detects that a lower driver has failed the IRP, the function or filter driver must undo any operations it performed in response to the IRP and propagate the error. If the function or filter driver propagated the usage information to any other device stacks, the driver must send another usage IRP to those stacks to notify them of the failure.

     If status is STATUS_SUCCESS and **InPath** is **TRUE**, clear the DO_POWER_PAGABLE bit.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Supporting Paging, Crash Dump, and Hibernation Files on a Device

When any of a driver's special file counts is nonzero, the driver must support the presence of the special file(s) on

its device (or a descendant device).

For a **DeviceUsageTypePaging** file created on its device, a driver must do the following:

- Lock code in memory for its *DispatchRead*, *DispatchWrite*, *DispatchDeviceControl*, and *DispatchPower* routines.
- Clear the DO_POWER_PAGABLE bit in its device object for the device (on the IRP's way up the device stack).
- Fail **IRP_MN_QUERY_STOP_DEVICE** and **IRP_MN_QUERY_REMOVE_DEVICE** requests for the device.

For a **DeviceUsageTypeDumpFile** file on its device, a driver must do the following:

- Lock code in memory for its *DispatchRead*, *DispatchWrite*, *DispatchDeviceControl*, and *DispatchPower* routines.
- Do not take the device out of the D0 state.

  Do not register the device for idle detection (**PoRegisterDeviceForIdleDetection**). If the device is already registered, cancel the registration. If the driver performs its own idle detection for the device, suspend such detection.
- Clear the DO_POWER_PAGABLE bit in its device object for the device (on the IRP's way up the device stack).
- Fail **IRP_MN_QUERY_STOP_DEVICE** and **IRP_MN_QUERY_REMOVE_DEVICE** requests for the device.

For a **DeviceUsageTypeHibernation** file on its device, a driver must do the following:

- Lock code in memory for its *DispatchRead*, *DispatchWrite*, *DispatchDeviceControl*, and *DispatchPower* routines.
- Ensure the device is in the D0 state when the driver receives an S4 system power IRP indicating that the system is about to hibernate.
- Do not power down the device in response to a D3 set-power IRP that is part of an S4 hibernate action. See System Power Actions for more information.

  Upon receipt of such a D3 set-power IRP, perform all tasks required to put the device in the D3 state except for powering off the device and notifying the power manager (**PoSetPowerState**). The device must retain power until the hibernation file has been written.
- Clear the DO_POWER_PAGABLE bit in its device object for the device (on the IRP's way up the device stack).
- Fail **IRP_MN_QUERY_STOP_DEVICE** and **IRP_MN_QUERY_REMOVE_DEVICE** requests for the device.

See Power Management for more information about device power states, power IRPs, and supporting power management in drivers.

Sending This IRP

A driver can send an **IRP_MN_DEVICE_USAGE_INFORMATION** IRP, but only to propagate device usage information to another device stack. A driver is never the initial source of device usage information.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DispatchDeviceControl

DispatchPower

DispatchRead

DispatchWrite

IoAdjustPagingPathCount

IoCallDriver

IoCompleteRequest

IO_STACK_LOCATION

IRP_MJ_PNP

IRP_MN_QUERY_DEVICE_RELATIONS

IRP_MN_QUERY_PNP_DEVICE_STATE

IRP_MN_QUERY_REMOVE_DEVICE

IRP_MN_QUERY_STOP_DEVICE

PoRegisterDeviceForIdleDetection

PoSetPowerState

IRP_MN_EJECT

6/25/2019 • 2 minutes to read • Edit Online

Bus drivers typically handle this request for their child devices (child PDOs) that support device ejection. Function and filter drivers do not receive this request.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP to direct the appropriate driver or drivers to eject the device from its slot.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

None

Output Parameters

None

I/O Status Block

A bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a bus driver sets **Irp->IoStatus.Information** to zero.

If a bus driver does not handle this IRP, it leaves **Irp->IoStatus.Status** as is and completes the IRP.

Operation

For the device to be ejected, the device must be in the D3 device power state (off) and must be unlocked (if the device supports locking).

Any driver that returns success for this IRP must wait until the device has been ejected before completing the IRP.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Instead, see the reference page for the **IoRequestDeviceEject** routine.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IoRequestDeviceEject

IRP_MN_FILTER_RESOURCE_REQUIREMENTS

6/25/2019 • 3 minutes to read • Edit Online

The PnP manager sends this IRP to a device stack so the function driver can adjust the resources required by the device, if appropriate.

The function driver typically handles this IRP.

The parent bus driver (and bus filter drivers) should not handle this request for a child PDO; instead, such a driver should report resource requirements in response to an **IRP_MN_QUERY_RESOURCE_REQUIREMENTS** request.

Upper and lower-filter drivers do not handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP when it is preparing to allocate resource(s) to a device.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in the context of an arbitrary thread.

Input Parameters

Irp->IoStatus.Information points to an **IO_RESOURCE_REQUIREMENTS_LIST** containing the hardware resource requirements for the device. The pointer is **NULL** if the device consumes no hardware resources.

Parameters.FilterResourceRequirements.IoResourceRequirementList also points to an **IO_RESOURCE_REQUIREMENTS_LIST**, but the function driver should use the list in the **IoStatus** block.

Output Parameters

Returned in the I/O status block.

I/O Status Block

If a function driver handles this IRP, it handles it on the IRP's way back up the stack. If the function driver handles the IRP successfully, it sets **Irp->IoStatus.Status** to STATUS_SUCCESS and sets **Irp->IoStatus.Information** to a pointer to an **IO_RESOURCE_REQUIREMENTS_LIST** containing the filtered resource requirements. See the "Operation" section below for more information about setting the filtered resource list. If a function driver encounters an error when handling this IRP, it sets the error in **Irp->IoStatus.Status**. If a function driver does not handle this IRP, it uses **IoSkipCurrentIrpStackLocation** to pass the IRP down the stack unchanged.

Upper and lower-filter drivers do not handle this IRP. Such a driver calls **IoSkipCurrentIrpStackLocation**, passes the IRP down to the next driver, must not modify **Irp->IoStatus**, and must not complete the IRP.

The parent bus driver does not handle this IRP. It leaves **Irp->IoStatus** as is and completes the IRP.

Operation

The PnP manager sends an **IRP_MN_QUERY_RESOURCE_REQUIREMENTS** request to the parent bus driver for the device, before the function driver has attached its device object to the device stack. To give the function

driver an opportunity to modify the device's resource requirements, if appropriate, the PnP manager later sends an **IRP_MN_FILTER_RESOURCE_REQUIREMENTS** request to the full device stack. The PnP manager sends this IRP before it allocates hardware resources to the device during initial device configuration. The PnP manager might also send this IRP during resource rebalancing.

When the PnP manager sends this IRP, it supplies the driver stack with a resource requirements list, which drivers can modify and return. The PnP manager supplies one of the following types of resource requirements list (listed in order of priority):

- Forced configuration (modified from a resource list to a resource requirements list)
- Override configuration
- Basic configuration
- Boot configuration (modified from a resource list to a resource requirements list)

If a function driver handles this IRP, it must set a completion routine and handle the IRP on its way back up the device stack. See Plug and Play for information about handling a PnP IRP on its way back up the device stack.

If the function driver is not changing the size of the current list pointed to by **Irp->IoStatus.Information**, the driver can modify the list in place. If the driver needs to change the size of the requirements list, the driver must allocate a new **IO_RESOURCE_REQUIREMENTS_LIST** list from paged memory and free the previous list. The PnP manager frees the returned structure when it is no longer needed.

A function driver must preserve the order of resources in the list pointed to by **Irp->IoStatus.Information** and must not alter resource tags that it does not handle. The driver must take care to adjust the requirements list in a way that the device's parent bus supports. If a function driver adds a new resource to the requirements list, and that resource is assigned to the device, the function driver should filter that resource out of the **IRP_MN_START_DEVICE** before passing the start IRP down to the bus driver.

If the function driver for the device does not handle this IRP, the PnP manager uses the resource requirements as specified by the parent bus driver in response to the **IRP_MN_QUERY_RESOURCE_REQUIREMENTS** request.

A function driver must be prepared to handle this IRP for a device at any time after the driver's *AddDevice* routine has been called for the device.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

ExAllocatePoolWithTag

ExFreePool

IO_RESOURCE_REQUIREMENTS_LIST

IRP_MN_START_DEVICE

IRP_MN_QUERY_BUS_INFORMATION

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager uses this IRP to request the type and instance number of a device's parent bus.

Bus drivers should handle this request for their child devices (PDOs). Function and filter drivers do not handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP when a device is enumerated.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

None

Output Parameters

Returned in the I/O status block.

I/O Status Block

A bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a bus driver sets **Irp->IoStatus.Information** to a pointer to a completed **PNP_BUS_INFORMATION** structure. (See the "Operation" section for more information.) On an error, the bus driver sets **Irp->IoStatus.Information** to zero.

Function and filter drivers do not handle this IRP.

Operation

The information returned in response to this IRP is available to the function and filter drivers for devices on the bus. Function and filter drivers can call **IoGetDeviceProperty** to request a **DevicePropertyBusTypeGuid**, **DevicePropertyLegacyBusType**, or **DevicePropertyBusNumber**. Function and filter drivers that support devices on more than one bus can use this information to determine on which bus a particular device resides.

If a bus driver returns information in response to this IRP, it allocates a **PNP_BUS_INFORMATION** structure from paged memory. The PnP manager frees the structure when it is no longer needed.

A **PNP_BUS_INFORMATION** structure has the following format:

```
typedef struct _PNP_BUS_INFORMATION {
    GUID BusTypeGuid;
    INTERFACE_TYPE LegacyBusType;
    ULONG BusNumber;
} PNP_BUS_INFORMATION, *PPNP_BUS_INFORMATION;
```

The members of the structure are defined as follows:

BusTypeGuid
A bus driver sets **BusTypeGuid** to the GUID for the type of the bus on which the device resides. GUIDs for standard bus types are listed in Wdmguid.h. Driver writers should generate GUIDs for other bus types using Uuidgen.

LegacyBusType
A PnP bus driver sets **LegacyBusType** to the **INTERFACE_TYPE** of the parent bus. The interface types are defined in Wdm.h. Some buses have a specific **INTERFACE_TYPE** value, such as **PCMCIABus**, **PCIBus**, or **PNPISABus**. For other buses, especially newer buses like USB, the bus driver sets this member to **PNPBus**.

The **LegacyBusType** specifies the interface used to communicate with the device. This may or may not correspond to the type of the parent bus. For example, the interface for a CardBus card that is plugged into a PCI CardBus controller is **PCIBus**. However, the interface for a PCMCIA card on a PCI CardBus controller is **PCMCIABus**.

BusNumber
A bus driver sets **BusNumber** to a number distinguishing the bus from other buses of the same type on the computer. The bus-numbering scheme is bus-specific. Bus numbers may be virtual, but must match any numbering used by legacy interfaces such as **IoReportResourceUsage**.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Call **IoGetDeviceProperty** to get information about the bus to which a device is attached.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IoGetDeviceProperty

IRP_MN_QUERY_CAPABILITIES

6/25/2019 • 4 minutes to read • Edit Online

The PnP manager sends this IRP to get the capabilities of a device, such as whether the device can be locked or ejected.

Function and filter drivers can handle this request if they alter the capabilities supported by the bus driver. Bus drivers must handle this request for their child devices.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP to the bus driver for a device immediately after the device is enumerated. The PnP manager sends this IRP again after all the drivers for a device have started the device. A driver can send this IRP to get the capabilities for a device.

The PnP manager and drivers send this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

The **Parameters.DeviceCapabilities.Capabilities** member of the **IO_STACK_LOCATION** structure points to a **DEVICE_CAPABILITIES** structure containing information about the capabilities of the device.

Output Parameters

Parameters.DeviceCapabilities.Capabilities points to the **DEVICE_CAPABILITIES** structure that reflects any modifications made by the drivers that handle the IRP.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_UNSUCCESSFUL.

If a function or filter driver does not handle this IRP, it calls **IoSkipCurrentIrpStackLocation** and passes the IRP down to the next driver. Such a driver must not modify **Irp->IoStatus.Status** and must not complete the IRP.

A bus driver sets **Irp->IoStatus.Status** and completes the IRP.

Operation

When a device is enumerated, but before the function and filter drivers are loaded for the device, the PnP manager sends an **IRP_MN_QUERY_CAPABILITIES** request to the parent bus driver for the device. The bus driver must set any relevant values in the **DEVICE_CAPABILITIES** structure and return it to the PnP manager.

After the device stack is built and drivers have started the device, the PnP manager sends this IRP again to be handled first by the driver at the top of the device stack and then by each lower driver in the stack. Function and filter drivers can set an *IoCompletion* routine and handle this IRP on its way back up the device stack.

Drivers should add capabilities before they pass the IRP to the next lower driver.

Drivers should remove capabilities after all lower drivers have finished with the IRP. A driver does not typically

remove capabilities that have been set by other drivers, but it might do so if it has special information about the capabilities of the device in a certain configuration. See Plug and Play for information about postponing IRP processing until lower drivers have finished.

After a device is enumerated and its drivers are loaded, its capabilities should not change. A device's capabilities might change if the device is removed and re-enumerated.

When handling an **IRP_MN_QUERY_CAPABILITIES** IRP, the driver that is the power policy manager for the device should set an *IoCompletion* routine and copy the device power capabilities, such as the S-to-D power state mappings, on the IRP's way back up the device stack. To determine the power capabilities of a child device, the parent bus driver creates another query-capabilities IRP and sends the IRP to its parent driver. See Reporting Device Power Capabilities for more information.

If a driver handles this IRP, it should check the **DEVICE_CAPABILITIES Version** value. If that value is not a version that the driver supports, the driver should fail the IRP. If the version is supported, the driver should check the **Size** field. A driver should set only those fields that are within the bounds of the capabilities structure that it received as input.

Drivers that handle this IRP can set some **DEVICE_CAPABILITIES** fields but must not set the **Size** and **Version** fields. These fields are only set by the component that sent the IRP.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

A bus driver sends this IRP to the parent device stack when it handles an **IRP_MN_QUERY_CAPABILITIES** request for one of its child devices. Also, a driver might send this IRP to get the device capabilities for one of its devices. A single driver in the stack has only part of the capabilities information for the device; sending an IRP to the device stack enables it to gather the full picture, including modifications by any filter drivers, and so forth.

See Handling IRPs for information about sending IRPs. The following steps apply specifically to this IRP:

- Allocate a **DEVICE_CAPABILITIES** structure from paged pool, and initialize it to zeros by calling **RtlZeroMemory**. Initialize the **Size** to **sizeof**(**DEVICE_CAPABILITIES**), the **Version** to 1, and **Address** and **UINumber** to -1.
- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to **IRP_MJ_PNP**, set **MinorFunction** to **IRP_MN_QUERY_CAPABILITIES**, and set **Parameters.DeviceCapabilities** to a pointer to the allocated **DEVICE_CAPABILITIES** structure.
- Initialize **IoStatus.Status** to STATUS_NOT_SUPPORTED.
- Deallocate the IRP and the **DEVICE_CAPABILITIES** structure when they are no longer needed.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DEVICE_CAPABILITIES

IRP_MN_QUERY_DEVICE_RELATIONS

6/28/2019 • 14 minutes to read • Edit Online

The PnP manager sends this request to determine certain relationships among devices. The following types of drivers handle this request:

- Bus drivers must handle **BusRelations** requests for their adapter or controller (bus FDO). Filter drivers might handle **BusRelations** requests.
- Bus drivers must handle **TargetDeviceRelation** requests for their child devices (child PDOs).
- Function and filter drivers might handle **RemovalRelations** and **PowerRelations** requests.
- Bus drivers might handle **EjectionRelations** requests for their child devices (child PDOs).

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP to gather information about devices with a relationship to the specified device.

The PnP manager queries a device's **BusRelations** (child devices) when the device is enumerated and at other times while the device is active, such as when a driver calls the **IoInvalidateDeviceRelations** routine to indicate that a child device has arrived or departed.

The PnP manager queries a device's **RemovalRelations** before it removes a device's drivers. The PnP manager queries for **RemovalRelations** and **EjectionRelations** before it ejects a device.

The PnP manager queries a device's **TargetDeviceRelation** when a driver or user-mode application registers for PnP notification of an **EventCategoryTargetDeviceChange** on the device. The PnP manager queries for the device that is associated with a particular file object. **IRP_MN_QUERY_DEVICE_RELATIONS** is the only PnP IRP that has a valid file object parameter. A driver can query a device stack for **TargetDeviceRelation**. A driver does not need to supply a file object when sending its **TargetDeviceRelation** query.

The PnP manager queries a device's **PowerRelations** when the driver for the device calls **IoInvalidateDeviceRelations** to indicate that the set of devices with which this device has an implicit power management relationship has changed. **PowerRelations** requests are supported starting with Windows 7.

For **BusRelations**, **RemovalRelations**, **EjectionRelations**, and **PowerRelations** requests, the PnP manager sends **IRP_MN_QUERY_DEVICE_RELATIONS** at IRQL = PASSIVE_LEVEL in the context of a system thread.

For **TargetDeviceRelation** requests, the PnP manager sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

The **Parameters.QueryDeviceRelations.Type** member of the **IO_STACK_LOCATION** structure specifies the type of relations that are being queried. Possible values include **BusRelations**, **EjectionRelations**, **RemovalRelations**, **TargetDeviceRelation**, and **PowerRelations**.

The **FileObject** member of the current **IO_STACK_LOCATION** structure points to a valid file object only if **Parameters.QueryDeviceRelations.Type** is **TargetDeviceRelation**.

Output Parameters

Returned in the I/O status block.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to a failure status such as STATUS_INSUFFICIENT_RESOURCES.

On success, a driver sets **Irp->IoStatus.Information** to a PDEVICE_RELATIONS pointer that points to the requested relations information. The **DEVICE_RELATIONS** structure is defined as follows:

```
typedef struct _DEVICE_RELATIONS {
  ULONG  Count;
  PDEVICE_OBJECT  Objects[1];  // variable length
} DEVICE_RELATIONS, *PDEVICE_RELATIONS;
```

Operation

If a driver returns relations in response to this **IRP_MN_QUERY_DEVICE_RELATIONS**, the driver allocates a **DEVICE_RELATIONS** structure from paged memory that contains a count and the appropriate number of device object pointers. The PnP manager frees the structure when it is no longer needed. If a driver replaces a **DEVICE_RELATIONS** structure that another driver allocated, the driver must free the previous structure.

A driver must reference the PDO of any device that it reports in this IRP (**ObReferenceObject**). The PnP manager removes the reference when appropriate.

A function or filter driver should be prepared to handle this IRP for a device any time after its *AddDevice* routine has completed for the device. Bus drivers should be prepared to handle a query for **BusRelations** immediately after a device is enumerated.

For the general rules about handling Plug and Play minor IRPs see Plug and Play.

The following subsections describe the specific actions for handling the various queries.

BusRelations Request

When the PnP manager queries for the bus relations (child devices) of an adapter or controller, the bus driver must return a list of pointers to the PDOs of any devices physically present on the bus. The bus driver reports all devices, regardless of whether they have been started. The bus driver might need to power up its bus device to determine which children are present.

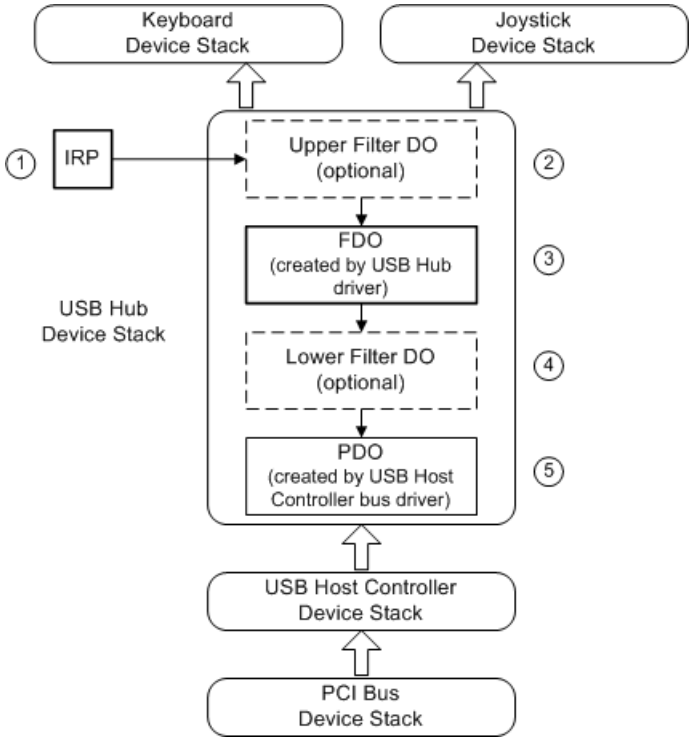
Warning A device object cannot be passed to any routine that takes a PDO as an argument until the PnP manager creates a device node (*devnode*) for that object. (If the driver does pass a device object, the system will bug check with **Bug Check 0xCA: PNP_DETECTED_FATAL_ERROR**.) The PnP manager creates the devnode in response to the **IRP_MN_QUERY_DEVICE_RELATIONS** request. The driver can safely assume that the PDO's devnode has been created when it receives an **IRP_MN_QUERY_RESOURCE_REQUIREMENTS** request.

The bus driver that responds to this IRP is the function driver for the bus adapter or controller, not the parent bus driver for the bus that the adapter or controller is connected to. Function drivers for non-bus devices do not handle this query. Such drivers just pass the IRP to the next lower driver. (See the following figure.) Filter drivers typically do not handle this query.

On Windows Vista and later operating systems, we recommend that drivers always pend the **IRP_MN_QUERY_DEVICE_RELATIONS** IRP and complete its processing later. This order enables the system to process bus relation queries asynchronously. (On operating systems before Windows Vista, drivers can safely return STATUS_PENDING from their dispatch routines, but the PnP manager does not overlap the bus relation

query with any other operation.)

The following diagram shows how drivers handle a query for bus relations.



In the example shown in the figure, the PnP manager sends an **IRP_MN_QUERY_DEVICE_RELATIONS** for **BusRelations** to the drivers for the USB hub device. The PnP manager is requesting a list of the hub device's children.

1. As with all PnP IRPs, the PnP manager sends the IRP to the top driver in the device stack for the device.
2. An optional filter driver might be the top driver in the stack. A filter driver typically does not handle this IRP; it passes the IRP down the stack. A filter driver might handle this IRP, for example, if the driver exposes a non-enumerable device on the bus.
3. The USB hub bus driver handles the IRP.

   The USB hub bus driver:

   - Creates a PDO for any child device that does not already have one.
   - Marks the PDO inactive for any device that is no longer present on the bus. The bus driver does not delete such PDOs.For more information about when to delete the PDOs, see Removing a Device.
   - Reports any child devices that are present on the bus.

     For each child device, the bus driver references the PDO and puts a pointer to the PDO in the DEVICE_RELATIONS structure.

     There are two PDOs in this example: one for the joystick device and one for the keyboard device.

     The bus driver should check whether another driver already created a DEVICE_RELATIONS structure for this IRP. If so, the bus driver must add to the existing information.

     If there is no child device present on the bus, the driver sets the count to zero in the DEVICE_RELATIONS structure and returns success.

   - Sets the appropriate values in the I/O status block and passes the IRP to the next lower driver. The bus driver for the adapter or controller does not complete the IRP.
4. An optional lower filter, if present, typically does not handle this IRP. Such a filter driver passes the IRP

down the stack. If a lower-filter driver handles this IRP, it can add PDO(s) to the list of child devices but it must not delete any PDOs created by other drivers.

5. The parent bus driver does not handle this IRP, unless it is the only driver in the device stack (the device is in raw mode). As with all PnP IRPs, the parent bus driver completes the IRP with **IoCompleteRequest**.

   If there are one or more bus filter drivers in the device stack, such drivers might handle the IRP on its way down to the bus driver and/or on the IRP's way back up the device stack (if there are *IoCompletion* routines). According to the PnP IRP rules, such a driver can add PDOs to the IRP on its way down the stack and/or modify the relations list on the IRP's way back up the stack (in *IoCompletion* routines).

EjectionRelations Request

A driver returns pointers to PDOs of any devices that might be physically removed from the system when the specified device is ejected. Do not report the PDOs of children of the device; the PnP manager always requests that child devices be removed before their parent device.

The PnP manager sends an **IRP_MN_EJECT** IRP to a device that is being ejected. The driver for such a device also receive a remove IRP. The device's ejection relations receive an **IRP_MN_REMOVE_DEVICE** IRP (not an **IRP_MN_EJECT** IRP).

Only a parent bus driver can respond to an **EjectionRelations** query for one of its child devices. Function and filter drivers must pass it to the next lower driver in the device stack. If a bus driver receives this IRP as the function driver for its adapter or controller, the bus driver is performing the tasks of a function driver and must pass the IRP to the next lower driver.

PowerRelations Request

Starting with Windows 7, the **PowerRelations** query enables a driver to specify a power management relationship outside of the conventional relationship between a parent bus that supports PnP enumeration and an enumerated child device on the bus. For example, if a bus driver cannot enumerate a child device on the bus, or if a device is a child of more than one bus, the **PowerRelations** query can describe the child device's power relations with the bus or buses.

The PnP manager issues a **PowerRelations** query for a device when the driver for the device calls the **IoInvalidateDeviceRelations** routine and specifies a *Type* parameter value of **PowerRelations**.

In response to this query, the driver for the target device (that is, the device that is the target for the query) supplies a **DEVICE_RELATIONS** structure that contains pointers to the PDOs of any other devices that must be turned on by the power manager before the target device is turned on. Conversely, these other devices must be turned off only after the target device is turned off. The power manager uses the information from the query to guarantee that these devices are turned on and off in the correct order.

This ordering guarantee applies only to global system sleep state transitions, which include transitions to and from the S1, S2, S3 (*sleep*), S4 (*hibernate*), and S5 (*shutdown*) system power states. The **PowerRelations** ordering guarantee does not apply to Dx device power state transitions while the system stays in the S0 (*running*) system state, except in the case of Directed Runtime Power Management (DFx) transitions.

If the target device is on the device path for a special file (such as the paging file, hibernate file, or crash dump file), the driver for the target device must perform an additional step when it handles an **IRP_MN_DEVICE_USAGE_NOTIFICATION** IRP in which **InPath** is **TRUE**. This driver must ensure that the devices whose PDOs are supplied for the **PowerRelations** query can also support being in the device path for the special file. To confirm this support, the driver for the target device must first send the **IRP_MN_DEVICE_USAGE_NOTIFICATION** IRP to each of these devices, and this IRP must specify the same **UsageNotification.Type** as the target device. Only if all the devices that receive this IRP complete the IRP with a success status code can the driver for the target device complete its **IRP_MN_DEVICE_USAGE_NOTIFICATION** IRP successfully. Otherwise, this driver must complete this IRP

with a failure status code.

When this same driver handles an **IRP_MN_DEVICE_USAGE_NOTIFICATION** IRP for which **InPath** is **FALSE**, the driver must send the **IRP_MN_DEVICE_USAGE_NOTIFICATION** IRP to the same set of dependent devices as for the case in which **InPath** is **TRUE**. However, the driver should never complete this IRP with a failure status code when **InPath** is **FALSE**.

The driver that responds to the **PowerRelations** query should register for target device change notifications on all devices whose PDOs are supplied for the **PowerRelations** query. To register for these notifications, the driver can call the **IoRegisterPlugPlayNotification** routine and specify an *EventCategory* parameter value of **EventCategoryTargetDeviceChange**.

RemovalRelations Request

A driver returns pointers to PDOs of any devices whose drivers must be removed when the drivers for the specified device are removed. Do not report the PDOs of children of the device; the PnP manager already requests the removal of child devices before removing a device.

The order in which removal relations are removed is undefined.

Any driver in the device stack can handle this type of relations query. A function or filter driver handles the IRP before passing it to the next lower driver. A bus driver handles the IRP and then completes it.

TargetDeviceRelation Request

The **TargetDeviceRelation** query enables the PnP manager to query a non-PnP device stack for the PDO in the PnP device stack that controls the hardware.

In general, drivers forward the **IRP_MN_QUERY_DEVICE_RELATIONS** IRP down their stack until the IRP reaches the bottom of a particular device stack. A driver at the bottom of a non-PnP stack then forwards or re-issues the IRP to the relevant PnP stack. For example, the PnP manager might send a **TargetDeviceRelation** query to the device object at the top of the file system stack, which is a non-PnP stack. Each device object in the file system stack would pass the query to the device object below it until the query reached the device object at the bottom of the stack. The lowest device object in the stack would forward or re-issue the **TargetDeviceRelation** query to the device object at the top of the PnP storage volume stack, and then the query would be passed down to the PDO at the bottom of the storage volume stack.

The following list summarizes the situations in which you can safely acquire a pointer to the PDO at the bottom of a PnP device stack:

- Device object in a PnP

  A device object that is in a PnP device stack learns about the stack's PDO when the *AddDevice* routine for the device is called. The driver can safely cache the pointer to the PDO if the use of the pointer is properly synchronized with incoming **IRP_MN_REMOVE_DEVICE** messages by using the remove lock routines.

- Device object in a non-PnP stack, not at bottom of stack

  For a device object that is not at the bottom of a non-PnP stack, a driver can send a **TargetDeviceRelation** query to obtain a pointer to the PDO at the bottom of the corresponding PnP device stack.

- File object for the device

  Given a file object for the device, a driver can call **IoGetRelatedDeviceObject** to get the device object and then follow the instructions in the preceding list item.

- Handle to the device object

  Given a handle to the device object, a driver can call **ObReferenceObjectByHandle** to get the file object for the device and then follow the instructions in the preceding list item.

A parent bus driver must handle a **TargetDeviceRelation** relations query for its child devices. The bus driver references the child device's PDO with **ObReferenceObject** and returns a pointer to the PDO in the **DEVICE_RELATIONS** structure. There is only one PDO pointer in the structure for this relation type. The PnP manager removes the reference to the PDO when the driver or application unregisters for notification on the device.

Only a parent bus driver responds to a **TargetDeviceRelation** query. Function and filter drivers must pass it to the next lower driver in the device stack. If a bus driver receives this IRP as the function driver for its adapter or controller, the bus driver is performing the tasks of a function driver and must pass the IRP to the next lower driver.

If a driver is not in a PDO-based stack, the driver sends a new target-device-relation query IRP to the device object associated with the file handle on which the driver performs I/O.

Sending This IRP

Drivers must not send **IRP_MN_QUERY_DEVICE_RELATIONS** to request **BusRelations**. Drivers are not restricted from sending this IRP for **RemovalRelations** or **EjectionRelations**, but it is not likely that a driver would do so.

Drivers can query a device stack for **TargetDeviceRelation**. See Handling IRPs for information about sending IRPs. The following steps apply specifically to this IRP:

- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to **IRP_MJ_PNP**, set **MinorFunction** to **IRP_MN_QUERY_DEVICE_RELATIONS**, set **Parameters.QueryDeviceRelations.Type** to **TargetDeviceRelation**, and set **Irp->FileObject** to a valid file object.
- Initialize **IoStatus.Status** to STATUS_NOT_SUPPORTED.

If a driver sent this IRP to get the PDO to report in response to an **IRP_MN_QUERY_DEVICE_RELATIONS** for **TargetDeviceRelation** that the driver received, then the driver reports the PDO and frees the returned relations structure when the IRP completes. If a driver initiated this IRP for another reason, the driver frees the relations structure when the IRP completes and dereferences the PDO when it is no longer needed.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

AddDevice

IoCompleteRequest

IoGetRelatedDeviceObject

IoInvalidateDeviceRelations

IoRegisterPlugPlayNotification

IRP_MJ_PNP

IRP_MN_DEVICE_USAGE_NOTIFICATION

IRP_MN_EJECT

IRP_MN_QUERY_RESOURCE_REQUIREMENTS

IRP_MN_REMOVE_DEVICE

IO_STACK_LOCATION

ObReferenceObject

ObReferenceObjectByHandle

IRP_MN_QUERY_DEVICE_TEXT

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager uses this IRP to get a device's description or location information.

Bus drivers must handle this request for their child devices if the bus supports this information. Function and filter drivers do not handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends two of these IRPs when a device is enumerated: one to query the device description and one to query the location information.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

The **Parameters.QueryDeviceText.DeviceTextType** member of the **IO_STACK_LOCATION** structure is a **DEVICE_TEXT_TYPE** value specifying which string is requested. Possible values for **DEVICE_TEXT_TYPE** include **DeviceTextDescription** and **DeviceTextLocationInformation**.

Parameters.QueryDeviceText.LocaleId is an LCID specifying the locale for the requested text.

Output Parameters

Returned in the I/O status block.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a bus driver sets **Irp->IoStatus.Information** to a pointer to a driver-allocated block of memory containing a WCHAR buffer with the requested information. On an error, the bus driver sets **Irp->IoStatus.Information** to zero.

Operation

Bus drivers are strongly encouraged to return device descriptions for their child devices. This string is displayed in the **Found New Hardware** pop-up window if no INF match is found for the device.

Bus drivers are also encouraged to return **LocationInformation** for their child devices, but this information is optional. The format of this string depends on the bus. The Device manager displays this string in the general properties tab for the device. Vendors should choose a string that conveys useful information to users and support personnel. For example, for PCI, the string contains the bus, device, and function. For PC Card, the string contains the slot.

If a bus driver returns information in response to this IRP, it allocates a NULL-terminated Unicode string from paged memory. The PnP manager frees the string when it is no longer needed.

If a device does not provide description or location information, the device's parent bus driver completes the IRP

(**IoCompleteRequest**) without modifying **Irp->IoStatus.Status** or **Irp->IoStatus.Information**.

Function and filter drivers do not handle this IRP; they pass it to the next lower driver with no changes to **Irp->IoStatus**.

Drivers for buses that support different text strings for different locales should be able to handle a request for a language that is not explicitly supported by the device. In such a situation, the bus driver should return the closest match for the locale or should fallback and return some appropriate supported locale string.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

IRP_MN_QUERY_ID

6/25/2019 • 7 minutes to read • Edit Online

Bus drivers must handle requests for **BusQueryDeviceID** for their child devices (child PDOs). Bus drivers can handle requests for **BusQueryHardwareIDs**, **BusQueryCompatibleIDs**, and **BusQueryInstanceID** for their child devices.

Beginning with Windows 7, bus drivers must also handle requests for BusQueryContainerID for their child PDOs.

For more information about these identifiers (IDs), see Device Identification Strings.

Note Function drivers and filter drivers do not handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP when a device is enumerated. A driver might send this IRP to retrieve the instance ID for one of its devices.

The PnP manager and drivers send this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

The **Parameters.QueryId.IdType** member of the **IO_STACK_LOCATION** structure specifies the kind of ID(s) requested. Possible values include BusQueryDeviceID, BusQueryHardwareIDs, BusQueryCompatibleIDs, BusQueryInstanceID, and BusQueryContainerID. The following ID type is reserved: BusQueryDeviceSerialNumber.

Output Parameters

Returned in the I/O status block.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a driver sets **Irp->IoStatus.Information** to a WCHAR pointer that points to the requested information. On error, a driver sets **Irp->IoStatus.Information** to zero.

Operation

If a driver returns ID(s) in response to this IRP, it allocates a WCHAR structure from paged pool to contain the ID(s). The PnP manager frees the structure when it is no longer needed.

A driver returns one of the following:

- A REG_SZ string in response to a BusQueryDeviceID, BusQueryInstanceID, or, BusQueryContainerID request.
- A REG_MULTI_SZ string in response to a BusQueryHardwareIDs or BusQueryCompatibleIDs request.

If a driver returns an ID with an illegal character, the system will bug check. Characters with the following values

are illegal in an ID for this IRP:

- Less than or equal to 0x20 (' ')
- Greater than 0x7F
- Equal to 0x2C (',')

A driver must conform to the following length restrictions for IDs:

- Each hardware ID or compatible ID that a driver returns in this IRP must be less than MAX_DEVICE_ID_LEN characters long. This constant currently has a value of 200 as defined in sdk\inc\cfgmgr32.h.
- The container ID that a driver returns in this IRP must be formatted as a globally unique identifier (GUID), and must be MAX_GUID_STRING_LEN characters, which includes the null terminator.
- If a bus driver supplies globally unique instance IDs for its child devices (that is, the driver sets **DEVICE_CAPABILITIES.UniqueID** for the devices), then the combination of device ID plus instance ID must be less than (MAX_DEVICE_ID_LEN - 1) characters. The operating system requires the additional character for a path separator.
- If a bus driver does not supply globally unique instance IDs for its child devices, then the combination of device ID plus instance ID must be less than (MAX_DEVICE_ID_LEN - 28). The value of this equation is currently 172.

Bus drivers should be prepared to handle this IRP for a child device immediately after the device is enumerated.

Specifying BusQueryDeviceID and BusQueryInstanceID

The values a bus driver supplies for BusQueryDeviceID and BusQueryInstanceID allow the operating system to differentiate a device from other devices on the computer. The operating system uses the device ID and instance ID that are returned in the **IRP_MN_QUERY_ID** IRP and the unique ID field that are returned in the **IRP_MN_QUERY_CAPABILITIES** IRP to locate registry information for the device.

For **BusQueryDeviceID**, a bus driver supplies the device's *device ID*. A device ID should contain the most-specific description of the device possible, incorporating the name of the enumerator and strings identifying the manufacturer, device, revision, packager, and packaged product, where possible. For example, the PCI bus driver responds with device IDs of the form PCI\VEN_xxxx&DEV_xxxx&SUBSYS_xxxxxxxx&REV_xx, encoding all five of the items mentioned above. However, a device ID should not contain enough information to differentiate between two identical devices. This information should be encoded in the instance ID.

For BusQueryInstanceID, a bus driver should supply a string that contains the *instance ID* for the device. Setup and bus drivers use the instance ID, with other information, to differentiate between two identical devices on the computer. The instance ID is either unique across the whole computer or just unique on the device's parent bus.

If an instance ID is only unique on the bus, the bus driver specifies that string for BusQueryInstanceID but also specifies a **UniqueID** value of **FALSE** in response to an **IRP_MN_QUERY_CAPABILITIES** request for the device. If **UniqueID** is **FALSE**, the PnP manager enhances the instance ID by adding information about the device's parent and thus makes the ID unique on the computer. In this case the bus driver should not take extra steps to make its devices' instance IDs globally unique; just return the appropriate capabilities information and the operating system takes care of it.

If a bus driver can supply a globally unique ID for each child device, such as a serial number, the bus driver specifies those strings for BusQueryInstanceID and specifies a **UniqueID** value of **TRUE** in response to an **IRP_MN_QUERY_CAPABILITIES** request for each device.

Specifying BusQueryHardwareIDs and BusQueryCompatibleIDs

The values a bus driver supplies for BusQueryHardwareIDs and BusQueryCompatibleIDs allow Setup to locate the appropriate drivers for the bus's child device.

A bus driver responds to each of these requests with a REG_MULTI_SZ list of IDs that describe the device. The maximum length, in characters, of a list of IDs, including the two NULL characters that terminate the list, is REGSTR_VAL_MAX_HCID_LEN.

When returning more than one *hardware ID* and/or more than one *compatible ID*, a bus driver should list the IDs in the order of most-specific to most-general to facilitate choosing the best driver match for the device. The first entry in the hardware IDs list is the most-specific description of the device and, as such, it is usually identical to the device ID.

Setup checks the IDs against the IDs listed in INF files for possible matches. Setup first scans the hardware IDs list, then the compatible IDs list. Earlier entries are treated as more specific descriptions of the device, and later entries as more general (and thus less optimal) matches for the device. If no match is found in the list of hardware IDs, Setup might prompt the user for installation media before moving on to the list of compatible IDs.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Specifying BusQueryContainerIDs

Beginning with Windows 7, a bus driver should supply a string for BusQueryContainerID that contains the container ID for the device. The container ID allows the operating system to group all functional devices from a single removable physical device. For example, all functional devices from a removable multifunction device have the same container ID. For more information about reporting container IDs in special cases, such as a volume device that may span multiple disks in multiple containers but does not belong to any container, see Overview of Container IDs.

A removable physical device is defined as a child device that the bus driver specifies a **Removable** capability of **TRUE** in response to an **IRP_MN_QUERY_CAPABILITIES** request. For more information about the **Removable** value, see **DEVICE_CAPABILITIES**.

The bus driver creates a container ID based on a bus-specific unique ID that the device provides. For more information, see How Container IDs are Generated.

The driver must fail the IRP request and set **IoStatus.Status** to STATUS_NOT_SUPPORTED if any of the following are true:

- The device does not support a bus-specific unique ID that the bus driver can use to generate a container ID.
- The bus driver had previously specified a **Removable** capability of **FALSE** in response to an **IRP_MN_QUERY_CAPABILITIES** request for the device.

Sending This IRP

Typically, only the PnP manager sends this IRP.

To get the hardware IDs or compatible IDs for a device, call **IoGetDeviceProperty** instead of sending this IRP.

A driver might send this IRP to retrieve the instance ID for one of its devices. For example, consider a multifunction PnP ISA device whose functions do not operate independently. The PnP manager enumerates the functions as separate devices, but the driver for such a device might be required to associate one or more of the functions. Because PnP ISA guarantees a unique instance ID, the driver for such a multifunction device can use the instance IDs to locate functions that reside on the same device. The driver for such a device must also get the device's enumerator name by calling **IoGetDeviceProperty**, to confirm that the device is a PnP ISA device.

See Handling IRPs for information about sending IRPs. The following steps apply specifically to this IRP:

- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to **IRP_MJ_PNP**, set

MinorFunction to IRP_MN_QUERY_ID, and set **Parameters.QueryId.IdType** to **BusQueryInstanceID**.

- Set **IoStatus.Status** to STATUS_NOT_SUPPORTED.

In addition to sending the query ID IRP, the driver must call **IoGetDeviceProperty** to get the **DevicePropertyEnumeratorName** for the device.

After the IRP completes and the driver is finished with the ID, the driver must free the ID structure returned by the driver(s) that handled the query IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

Device Identification Strings

IoGetDeviceProperty

IRP_MN_QUERY_INTERFACE

6/25/2019 • 6 minutes to read • Edit Online

The **IRP_MN_QUERY_INTERFACE** request enables a driver to export a direct-call interface to other drivers.

A bus driver that exports an interface must handle this request for its child devices (child PDOs). Function and filter can optionally handle this request.

An "interface" in this context consists of one or more routines, and possibly data, exported by a driver or set of drivers. An interface has a structure that describes its contents and a GUID that identifies its type.

For example, the PCMCIA bus driver exports an interface of type GUID_PCMCIA_INTERFACE_STANDARD that contains routines for operations such as getting the write-protect condition of a PCMCIA memory card. The function driver for such a memory card can send an **IRP_MN_QUERY_INTERFACE** request to the parent PCMCIA bus driver to get pointers to the PCMCIA interface routines.

This section describes the query-interface IRP as a general mechanism. Drivers that expose an interface should provide additional information about their specific interface.

Major Code

IRP_MJ_PNP When Sent

A driver or system component sends this IRP to get information about an interface exported by a driver for a device.

A driver or system component sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

A driver can receive this IRP at any time after the driver's *AddDevice* routine has been called for the device. The device might or might not be started when this IRP is sent (that is, you cannot assume that the driver has successfully completed an **IRP_MN_START_DEVICE** request for the device).

Input Parameters

The **Parameters.QueryInterface** member of the **IO_STACK_LOCATION** structure is itself a structure, which describes the interface being requested. The structure contains the following information:

```
CONST GUID *InterfaceType;
USHORT Size;
USHORT Version;
PINTERFACE Interface;
PVOID InterfaceSpecificData
```

The members of the structure are defined as follows:

InterfaceType
Points to a GUID that identifies the interface being requested. The GUID can be for a system-defined interface, such as GUID_BUS_INTERFACE_STANDARD, or a custom interface. The GUIDs for system-defined interfaces are listed in Wdmguid.h. GUIDs for custom interfaces should be generated with Uuidgen.

Size
Specifies the size of the interface being requested. Drivers that handle this IRP must not return an **INTERFACE** structure larger than **Size** bytes.

Version
Specifies the version of the interface being requested.

If a driver supports more than one version of an interface, the driver returns the closest supported version without exceeding the requested version. The component that sent the IRP should examine the returned **Interface.Version** field and determine what to do based on that value.

Interface
Points to a structure in which to return the requested interface. This structure must contain an **INTERFACE** structure as its first member. The component sending the IRP allocates this structure from paged memory.

A driver that exports an interface defines a new structure type containing the **INTERFACE** structure, plus members for routines and/or data in the interface. (The driver also defines a GUID for the interface, as described in the **InterfaceType** member, above.)

A driver that exports an interface defines the execution environment for each routine in the interface, including the IRQL at which the routine can be called, and so forth.

InterfaceSpecificData
Specifies additional information about the interface being requested.

For some interfaces, the component sending the IRP specifies additional information in this field. Typically, this field is **NULL** and the **InterfaceType** and **Version** are sufficient to identify the interface being requested.

Output Parameters

On success, a driver fills in the members of the **Parameters.QueryInterface.Interface** structure.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a bus driver sets **Irp->IoStatus.Information** to zero.

If a function or filter driver does not handle this IRP, it calls **IoSkipCurrentIrpStackLocation** and passes the IRP down to the next driver. Such a driver must not modify **Irp->IoStatus.Status** and must not complete the IRP.

If a bus driver does not export the requested interface and therefore does not handle this IRP for a child PDO, the bus driver leaves **Irp->IoStatus.Status** as is and completes the IRP.

Operation

A driver handles this IRP if the parameters specify an interface the driver supports.

A driver must not queue this IRP if the IRP requests an interface that the driver does not support. A driver must check **Parameters.QueryInterface.InterfaceType** in its **IO_STACK_LOCATION** structure. If the interface is not one the driver supports, the driver must pass the IRP to the next lower driver in the device stack without blocking.

Each interface must provide **InterfaceReference** and **InterfaceDereference** routines, and the driver that exports the interface must supply the addresses of these routines in the **INTERFACE** structure. Before a driver returns an interface in response to the IRP, it must increment the interface's reference count by calling its **InterfaceReference** routine. When the driver that requested the interface has finished using it, that driver must decrement the reference count by calling the interface's **InterfaceDereference** routine.

If the driver that sends the IRP (driver *x*) later passes the interface to another driver (driver *y*) then driver *x* must increment the interface's reference count and driver *y* must decrement it.

A driver that handles this IRP should avoid passing the IRP to another device stack to get the requested interface.

Such a design would create dependencies between the device stacks that are difficult to manage. For example, the device represented by the second device stack cannot be removed until the appropriate driver in the first stack dereferences the interface.

Interfaces can be bus-specific or bus-independent. Bus-specific interfaces are defined in the header files for those buses. The system defines a bus-independent interface, **BUS_INTERFACE_STANDARD**, for exporting standard bus interfaces.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

This IRP is used specifically to pass routine entry points between layered kernel-mode drivers for a device. Do not confuse the interfaces exposed by this IRP with *device interfaces*. A device interface is used primarily for exposing a path to a device for use by user-mode components or other kernel components. For more information about device interfaces, see Device Interface Classes.

Sending This IRP

See Handling IRPs for information about sending IRPs. The following steps apply specifically to this IRP:

- Allocate an **INTERFACE** structure from paged pool and initialize it to zeros. If the interface will be called at IRQL >= DISPATCH_LEVEL, based on the interface contract, the caller can copy the contents to memory that is allocated from nonpaged pool.
- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to **IRP_MJ_PNP**, set **MinorFunction** to **IRP_MN_QUERY_INTERFACE**, and set the appropriate values in **Parameters.QueryInterface**.
- Initialize **IoStatus.Status** to STATUS_NOT_SUPPORTED.
- Deallocate the IRP and the **INTERFACE** structure when they are no longer needed.
- Use the interface routines and context parameter as described in the specification for the interface.
- Decrement the reference count using the *InterfaceDereference* routine when the interface is no longer needed. Do not call any interface routines after dereferencing the interface.

A driver typically sends this IRP to the top of the device stack in which the driver is attached. If a driver sends this IRP to a different device stack, the driver must register for target device notification on the other device if the other device is not an ancestor of the device that the driver is servicing. Such a driver calls **IoRegisterPlugPlayNotification** with an *EventCategory* of **EventCategoryTargetDeviceChange**. When the driver receives notification of type GUID_TARGET_DEVICE_QUERY_REMOVE, the driver must dereference the interface. The driver can requery for the interface if it receives a subsequent GUID_TARGET_DEVICE_REMOVE_CANCELLED notification.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

BUS_INTERFACE_STANDARD

INTERFACE

IoRegisterPlugPlayNotification

IRP_MN_QUERY_LEGACY_BUS_INFORMATION

12/5/2018 • 2 minutes to read • Edit Online

This IRP is reserved for system use.

Major Code

IRP_MJ_PNP Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

IRP_MN_QUERY_PNP_DEVICE_STATE

6/25/2019 • 2 minutes to read • Edit Online

Function, filter, and bus drivers can handle this request.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP after the drivers for a device return success from the **IRP_MN_START_DEVICE** request sent when a device is first started. This IRP is not sent on a start after a stop for resource rebalancing. The PnP manager also sends this IRP when a driver for the device calls **IoInvalidateDeviceState**.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in the context of an arbitrary thread.

Input Parameters

None

Output Parameters

Returned in I/O status block.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_UNSUCCESSFUL.

On success, a driver sets **Irp->IoStatus.Information** to a **PNP_DEVICE_STATE** bitmask.

If a function or filter driver does not handle this IRP, it calls **IoSkipCurrentIrpStackLocation**, does not set an *IoCompletion* routine, and passes the IRP down to the next driver. Such a driver must not modify **Irp->IoStatus** and must not complete the IRP.

If a bus driver does not handle this IRP, it leaves **Irp->IoStatus.Status** as is and completes the IRP.

Operation

This IRP is handled first by the driver at the top of the device stack and then by each next lower driver in the stack.

A driver handles this IRP if it has information about the PnP state of a device. A driver can set or clear the flags in the PNP_DEVICE_STATE bitmask. If another driver has set a PNP_DEVICE_STATE in **Irp->IoStatus.Information**, a driver must take care to modify the flags in that bitmask rather than overwrite the whole structure.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IoInvalidateDeviceState

PNP_DEVICE_STATE

IRP_MN_QUERY_REMOVE_DEVICE

6/25/2019 • 2 minutes to read • Edit Online

All PnP drivers must handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP to inform drivers that a device is about to be removed from the computer and to query whether the device can be removed without disrupting the computer. The PnP manager also sends this IRP if a user requests to update driver(s) for the device.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input Parameters

None

Output Parameters

None

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_UNSUCCESSFUL.

Operation

This IRP is handled first by the driver at the top of the device stack and then passed down to each lower driver in the stack.

In response to this IRP, drivers indicate whether the device can be removed without disrupting the computer.

For more information about handling this IRP, see Handling an IRP_MN_QUERY_REMOVE_DEVICE Request. For general information about supporting device removal, see Removing a Device.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IRP_MN_CANCEL_REMOVE_DEVICE

IRP_MN_DEVICE_USAGE_NOTIFICATION

IRP_MN_REMOVE_DEVICE

IRP_MN_QUERY_RESOURCE_REQUIREMENTS

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager uses this IRP to get a device's resource requirements list.

Bus drivers must handle this request for their child devices that require hardware resources. Bus filter drivers can handle this request. Function and filter drivers do not handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP when a device is enumerated, prior to allocating resources to a device, and when a driver reports that its device's resource requirements have changed.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

None

Output Parameters

Returned in the I/O status block.

I/O Status Block

A driver that handles this IRP sets **Irp->IoStatus.Status** to STATUS_SUCCESS or an appropriate error status.

On success, a driver sets **Irp->IoStatus.Information** to a pointer to an **IO_RESOURCE_REQUIREMENTS_LIST** that contains the requested information. On an error, the driver sets **Irp->IoStatus.Information** to zero.

Operation

If a bus driver returns a resource requirements list in response to this IRP, it allocates an **IO_RESOURCE_REQUIREMENTS_LIST** from paged memory. The PnP manager frees the buffer when it is no longer needed.

If a device requires no hardware resources, the device's bus driver completes the IRP (**IoCompleteRequest**) without modifying **Irp->IoStatus.Status** or **Irp->IoStatus.Information**.

If a bus filter driver handles this IRP, it modifies the resource requirements list created by the bus driver. A bus filter driver modifies the list on the IRP's way back up the device stack. A bus filter driver must preserve the order of resources in the resource requirements list and must not alter resource tags that it does not handle. If a bus filter driver changes the size of the resource requirements list, the driver must allocate a new structure from paged memory and free the previous structure. If a bus filter driver adds a new resource requirement to the list and the resource is assigned to the device, the driver must filter the new resource out of the **IRP_MN_START_DEVICE** IRP so it is not passed to the bus driver.

Function and non-bus filter drivers do not handle this IRP; they pass it to the next lower driver with no changes to **Irp->IoStatus**.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IO_RESOURCE_REQUIREMENTS_LIST

IRP_MN_QUERY_RESOURCES

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager uses this IRP to get a device's boot configuration resources.

Bus drivers must handle this request for their child devices that require hardware resources. Function and filter drivers do not handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP when a device is enumerated.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

None

Output Parameters

Returned in the I/O status block.

I/O Status Block

A bus driver that handles this IRP sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a bus driver sets **Irp->IoStatus.Information** to a pointer to a **CM_RESOURCE_LIST** that contains the requested information. On an error, the bus driver sets **Irp->IoStatus.Information** to zero.

Operation

If a bus driver returns a resource list in response to this IRP, it allocates a **CM_RESOURCE_LIST** from paged memory. The PnP manager frees the buffer when it is no longer needed.

If a device requires no hardware resources, the device's parent bus driver completes the IRP (**IoCompleteRequest**) without modifying **Irp->IoStatus.Status** or **Irp->IoStatus.Information**.

Function and filter drivers do not receive this IRP.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Drivers can call **IoGetDeviceProperty** to get the boot configuration for a device, in both raw and translated forms.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

CM_RESOURCE_LIST

IoGetDeviceProperty

IRP_MN_QUERY_STOP_DEVICE

6/25/2019 • 2 minutes to read • Edit Online

All PnP drivers must handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP to query whether a device can be stopped to rebalance resources.

On Windows 98/Me, the PnP manager also sends this IRP when a device is being disabled.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input Parameters

None

Output Parameters

None

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status. If a driver cannot stop the device, the driver sets **Irp->IoStatus.Status** to STATUS_UNSUCCESSFUL.

A bus driver can set **Irp->IoStatus.Status** to STATUS_RESOURCE_REQUIREMENTS_CHANGED to indicate success for the IRP but also to request that the PnP manager requery the resource requirements for the device before sending the stop IRP.

Operation

This IRP is handled first by the driver at the top of the device stack and then passed down to each lower driver in the stack.

In response to this IRP, Windows 2000 and later drivers indicate whether it is safe to stop the device for resource rebalancing.

On Windows 98/Me, this IRP is sent not only during resource rebalancing, but also when a device is being disabled. Because a driver cannot distinguish these two situations, it should proceed as if the device were being disabled. If there are any open handles to the device, the driver should fail this IRP. If no handles are open, the driver should proceed as described in Handling an IRP_MN_QUERY_STOP_DEVICE Request (Windows 98/Me).

See Plug and Play for the general rules for handling Plug and Play Minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IRP_MN_CANCEL_STOP_DEVICE

IRP_MN_DEVICE_USAGE_NOTIFICATION

IRP_MN_START_DEVICE

IRP_MN_STOP_DEVICE

IRP_MN_READ_CONFIG

6/25/2019 • 3 minutes to read • Edit Online

Bus drivers for buses with configuration space must handle this request for their child devices (child PDOs). Filter and function drivers do not handle this request.

Major Code

IRP_MJ_PNP When Sent

A driver or other system component sends this IRP to read the configuration space of a device's parent bus.

A driver or other system component sends this IRP at IRQL < DISPATCH_LEVEL in an arbitrary thread context.

Input Parameters

The **Parameters.ReadWriteConfig** member of the **IO_STACK_LOCATION** structure is itself a structure containing the following information:

```
ULONG WhichSpace;
PVOID Buffer;
ULONG Offset;
ULONG Length
```

The members of the structure can be interpreted differently by different bus drivers, but the members are typically defined as follows:

WhichSpace
Specifies which memory area to access. This parameter can take the following values:

VALUE	BUS	MEANING
PCI_WHICHSPACE_CONFIG	PCI	PCI configuration space.
PCI_WHICHSPACE_ROM	PCI	Read-only memory.
PCCARD_COMMON_MEMORY PCCARD_COMMON_MEMORY_INDIRECT	PCMCIA	Main PCCARD memory.
PCCARD_ATTRIBUTE_MEMORY PCCARD_ATTRIBUTE_MEMORY_INDIRECT	PCMCIA	PCMCIA attribute (configuration) space.
PCCARD_PCI_CONFIGURATION_SPACE	PCMCIA	PCI configuration space.

The PCI_*XXX* values are defined in Wdm.h. The PCCARD_*XXX* values are defined in Ntddpcm.h.

Buffer
Points to a buffer in which to return the requested information. The component sending the IRP allocates this structure from paged memory. The format of the buffer is bus-specific.

Offset
Specifies an offset into the configuration space.

Length
Specifies the number of bytes to read.

Output Parameters

On success, a bus driver fills the buffer at **Parameters.ReadWriteConfig.Buffer** with the requested data.

I/O Status Block

A bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_INVALID_PARAMETER_*n*, STATUS_NO_SUCH_DEVICE, or STATUS_DEVICE_NOT_READY.

On success, a bus driver sets **Irp->IoStatus.Information** to the number of bytes returned.

If a bus driver is unable to complete this request immediately it can mark the IRP pending, return STATUS_PENDING, and complete the IRP at a later time.

Operation

A bus driver handles this IRP for its child devices (child PDOs).

Function and filter drivers do not handle this IRP; they pass it to the next lower driver with no changes to **Irp->IoStatus**.Status and they do not set an *IoCompletion* routine.

A bus driver that handles this request should check the WhichSpace parameter to ensure that it contains a value that the driver supports.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

Typically, a function driver sends this IRP to the top driver in the device stack to which it is attached and the IRP is handled by the parent bus driver.

See Handling IRPs for information about sending IRPs. The following steps apply specifically to this IRP:

- Allocate a buffer from a paged pool and initialize it to zeros.
- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to **IRP_MJ_PNP**, set **MinorFunction** to **IRP_MN_READ_CONFIG**, and set the appropriate values in **Parameters.ReadWriteConfig**.
- Initialize **IoStatus.Status** to STATUS_NOT_SUPPORTED.
- Deallocate the IRP and the buffer when they are no longer needed.

Drivers must send this IRP from IRQL < DISPATCH_LEVEL.

A driver can access a bus's configuration space at DISPATCH_LEVEL through a bus interface routine, if the parent bus driver supports such an interface. To get a bus interface, a driver sends an **IRP_MN_QUERY_INTERFACE** request to the device stack in which the driver is attached. The driver then calls the appropriate routine returned in

the interface.

For example, to read configuration space from DISPATCH_LEVEL, a driver can call **IRP_MN_QUERY_INTERFACE** during driver initialization to get the **BUS_INTERFACE_STANDARD** interface from the parent bus driver. The driver sends the query IRP from IRQL PASSIVE_LEVEL. Later, from code at IRQL DISPATCH_LEVEL, the driver calls the appropriate routine returned in the interface, such as the **Interface.GetBusData** routine.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IRP_MN_QUERY_INTERFACE

IRP_MN_WRITE_CONFIG

IRP_MN_REMOVE_DEVICE

6/25/2019 • 2 minutes to read • Edit Online

All PnP drivers must handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager uses this IRP to direct drivers to remove a device's software representation (device objects, and so forth). The PnP manager sends this IRP when a device has been removed in an orderly fashion (for example, initiated by a user in the Unplug or Eject Hardware program), by surprise (a user pulls the device from its slot without prior warning), or when the user requests to update driver(s).

On Windows 2000 and later systems, the PnP manager also sends this IRP if one of the drivers in the device stack fails an **IRP_MN_START_DEVICE** request for the device.

For an orderly device removal, the PnP manager sends an **IRP_MN_QUERY_REMOVE_DEVICE** prior to the remove IRP. In this case, the device is in the remove-pending state when the remove IRP arrives. For a surprise device removal on Microsoft Windows 2000 or later, the PnP manager sends an **IRP_MN_SURPRISE_REMOVAL** prior to the remove IRP. In this case, the device is in the surprise-removed state when the remove IRP arrives. Drivers can also receive a remove IRP before a device is started. In this case, the device is in the non-started state when the IRP arrives.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input Parameters

None

Output Parameters

None

I/O Status Block

A driver must set **Irp->IoStatus.Status** to STATUS_SUCCESS. Drivers must not fail this IRP.

Operation

This IRP is handled first by the driver at the top of the device stack and then by each lower driver in the stack.

In response to this IRP, drivers perform such tasks as powering down the device, removing the device's software representation (device objects, and so forth), and releasing any resources for the device.

For more information about handling this IRP, see Handling an IRP_MN_REMOVE_DEVICE Request. For general information about supporting device removal, see Removing a Device.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

If a bus driver detects that one (or more) of its child devices (child PDOs) has been physically removed from the

computer, the bus driver calls **IoInvalidateDeviceRelations** to report the change to the PnP manager. The PnP manager then sends remove IRPs for any devices that have disappeared.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IoInvalidateDeviceRelations

IoRegisterPlugPlayNotification

IRP_MN_CANCEL_REMOVE_DEVICE

IRP_MN_QUERY_REMOVE_DEVICE

IRP_MN_SURPRISE_REMOVAL

IRP_MN_SET_LOCK

6/25/2019 • 2 minutes to read • Edit Online

Bus drivers must handle this IRP for their child devices (child PDOs) that support device locking. Function and filter drivers do not handle this request.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP to direct driver(s) to lock the device and prevent device eject, or to unlock the device.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

The **Parameters.SetLock.Lock** member of the **IO_STACK_LOCATION** structure is a BOOLEAN value specifying whether to lock (TRUE) or unlock (FALSE) the device.

Output Parameters

None

I/O Status Block

A bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a driver sets **Irp->IoStatus.Information** to zero.

If a bus driver does not handle this IRP, it leaves **Irp->IoStatus.Status** as is and completes the IRP.

Function and filter drivers do not handle this IRP. Such drivers call **IoSkipCurrentIrpStackLocation** and pass the IRP down to the next driver. Function and filter drivers do not set an *IoCompletion* routine, do not modify **Irp->IoStatus**, and must not complete the IRP.

Operation

If a driver returns success for this IRP, it ensures that the device has been locked or unlocked before completing the IRP.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

IRP_MN_START_DEVICE

6/25/2019 • 2 minutes to read • Edit Online

All PnP drivers must handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP after it has assigned hardware resources, if any, to the device. The device may have been recently enumerated and is being started for the first time, or the device may be restarting after being stopped for resource rebalancing.

Sometimes the PnP manager sends an **IRP_MN_START_DEVICE** to a device that is already started, supplying a different set of resources than the device is currently using. A driver initiates this action by calling **IoInvalidateDeviceState** and responding to the subsequent **IRP_MN_QUERY_PNP_DEVICE_STATE** request with the PNP_RESOURCE_REQUIREMENTS_CHANGED flag set. A bus driver might use this mechanism, for example, to open a new aperture on a PCI-to-PCI bridge.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input Parameters

The **Parameters.StartDevice.AllocatedResources** member of the **IO_STACK_LOCATION** structure points to a **CM_RESOURCE_LIST** describing the hardware resources that the PnP manager assigned to the device. This list contains the resources in raw form. Use the raw resources to program the device.

Parameters.StartDevice.AllocatedResourcesTranslated points to a **CM_RESOURCE_LIST** describing the hardware resources that the PnP manager assigned to the device. This list contains the resources in translated form. Use the translated resources to connect the interrupt vector, map I/O space, and map memory.

Output Parameters

None

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_UNSUCCESSFUL or STATUS_INSUFFICIENT_RESOURCES.

If a driver requires some time to run its start operations for a device, it can mark the IRP pending and return STATUS_PENDING.

Operation

This IRP must be handled first by the parent bus driver for a device and then by each higher driver in the device stack.

In response to this IRP, drivers start a device for the first time or restart a device that was stopped. The exact operations required to start a device vary from device to device, but can include powering on the device, performing device-specific initialization, and connecting the interrupt.

A driver can typically handle this IRP in the same way whether it is starting a device for the first time or restarting a device after an **IRP_MN_STOP_DEVICE**, except if a driver needs to restore device state on a restart after a stop.

On Windows Vista and later operating systems, we recommend that drivers always pend the **IRP_MN_START_DEVICE** IRP and complete its processing later. This order enables the system to process device restarts asynchronously. (On operating systems before Windows Vista, drivers can return STATUS_PENDING from their dispatch routines, but the PnP manager does not overlap the device restart with any other operation.)

For more information about handling a start IRP, see Starting a Device.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IRP_MN_STOP_DEVICE

IRP_MN_STOP_DEVICE

6/25/2019 • 2 minutes to read • Edit Online

All PnP drivers must handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP to stop a device so it can reconfigure the device's hardware resources.

On Windows 2000 and later systems, the PnP manager sends this IRP only if a prior **IRP_MN_QUERY_STOP_DEVICE** completed successfully.

On Windows 98/Me, the PnP manager also sends this IRP when a device is being disabled and when a device stack has failed an **IRP_MN_START_DEVICE** request. In cases of failed start, the PnP manager sends this IRP without a preceding **IRP_MN_QUERY_STOP_DEVICE** request.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input Parameters

None

Output Parameters

None

I/O Status Block

A driver must set **Irp->IoStatus.Status** to STATUS_SUCCESS.

Operation

This IRP is handled first by the driver at the top of the device stack and then passed down to each lower driver in the stack.

In response to this IRP, Windows 2000 and later drivers stop the device and release any hardware resources being used by the device, such as I/O ports and interrupts.

On Windows 2000 and later, a stop IRP is used solely to free a device's hardware resources so they can be reconfigured. Once the resources are reconfigured, the device is restarted. A stop IRP is not a precursor to a remove IRP. See Plug and Play for more information about the order in which PnP IRPs are sent to devices.

On Windows 98/Me, a stop IRP is also used after a failed start and when a device is being disabled. WDM drivers that run on these operating systems should stop the device, fail any incoming I/O, and disable and deregister any user-mode interfaces.

A driver must not fail this IRP. If a driver cannot release the device's hardware resources, it must fail the preceding query-stop IRP.

See Stopping a Device for detailed information about handling stop IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IRP_MN_QUERY_STOP_DEVICE

IRP_MN_START_DEVICE

IoSetDeviceInterfaceState

IoRegisterDeviceInterface

IRP_MN_SURPRISE_REMOVAL

6/25/2019 • 2 minutes to read • Edit Online

All PnP drivers must handle this IRP.

Major Code

IRP_MJ_PNP When Sent

The PnP manager sends this IRP to notify the drivers for a device that the device is no longer available for I/O operations. This IRP is sent on Windows 2000 and later systems only.

The PnP manager sends this IRP before notifying user-mode applications or other kernel-mode components. After this IRP completes, the PnP manager notifies registered applications and drivers that the device has been removed.

The device can be in any PnP state when the PnP manager sends this IRP.

On Windows 98/Windows Me, the PnP manager does not send this IRP.

The PnP manager sends this IRP at IRQL = PASSIVE_LEVEL in the context of a system thread.

Input Parameters

None

Output Parameters

None

I/O Status Block

A driver must set **Irp->IoStatus.Status** to STATUS_SUCCESS. A driver must not fail this IRP.

Operation

This IRP is handled first by the driver at the top of the device stack and then passed down to each lower driver in the stack.

For more information about this IRP, see Handling an IRP_MN_SURPRISE_REMOVAL Request. For additional information about supporting device removal, see Removing a Device.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IRP_MN_REMOVE_DEVICE

IRP_MN_WRITE_CONFIG

6/25/2019 • 2 minutes to read • Edit Online

Bus drivers for buses with configuration space must handle this request for their child devices (child PDOs). Function and filter drivers do not handle this request.

Major Code

IRP_MJ_PNP When Sent

A driver or other system component sends this IRP to write data to the configuration space of a device's parent bus.

A driver or other system component sends this IRP at IRQL < DISPATCH_LEVEL in an arbitrary thread context.

Input Parameters

Parameters.ReadWriteConfig is a structure containing the following information:

```
ULONG WhichSpace;
PVOID Buffer;
ULONG Offset;
ULONG Length
```

The members of the structure can be interpreted differently by different bus drivers, but the members are typically defined as follows:

WhichSpace
Specifies the configuration space. For information about values that can be specified for **WhichSpace**, see **IRP_MN_READ_CONFIG**.

Buffer
Points to a buffer that contains the data to be written. The format of the buffer is bus-specific.

Offset
Specifies an offset into the configuration space.

Length
Specifies the number of bytes to be written.

Output Parameters

Returned in the I/O status block.

I/O Status Block

A bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_INVALID_PARAMETER_*n*, STATUS_NO_SUCH_DEVICE, or STATUS_DEVICE_NOT_READY.

On success, a bus driver sets **Irp->IoStatus.Information** to the number of bytes written.

If a bus driver is unable to complete this request immediately, it can mark the IRP pending, return

STATUS_PENDING, and complete the IRP at a later time.

Operation

A bus driver handles this IRP for its child devices (child PDOs).

Function and filter drivers do not handle this IRP; they pass it to the next lower driver with no changes to **Irp->IoStatus.Status** and do not set an *IoCompletion* routine.

See Plug and Play for the general rules for handling Plug and Play minor IRPs.

Sending This IRP

Typically, a function driver sends this IRP to the device stack to which it is attached and the IRP is handled by the parent bus driver.

See Handling IRPs for information about sending IRPs. The following steps apply specifically to this IRP:

- Allocate a buffer from paged pool and initialize it with the data to be written.
- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to **IRP_MJ_PNP**, set **MinorFunction** to **IRP_MN_WRITE_CONFIG**, and set the appropriate values in **Parameters.ReadWriteConfig**.
- Initialize **IoStatus.Status** to STATUS_NOT_SUPPORTED.
- Deallocate the IRP and the buffer when they are no longer needed.

Drivers must send this IRP from IRQL < DISPATCH_LEVEL.

A driver can access a bus's configuration space at DISPATCH_LEVEL through a bus interface routine, if the parent bus driver exports such an interface. To get a bus interface, a driver sends an **IRP_MN_QUERY_INTERFACE** request to its parent bus driver. The driver then calls the appropriate routine returned in the interface.

For example, to write configuration space from DISPATCH_LEVEL a driver can call **IRP_MN_QUERY_INTERFACE** during driver initialization to get the **BUS_INTERFACE_STANDARD** interface from the parent bus driver. The driver sends the query IRP from IRQL PASSIVE_LEVEL. Later, from code at IRQL DISPATCH_LEVEL, the driver calls the appropriate routine returned in the interface, such as the **Interface.SetBusData** routine.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IRP_MN_QUERY_INTERFACE

IRP_MN_READ_CONFIG

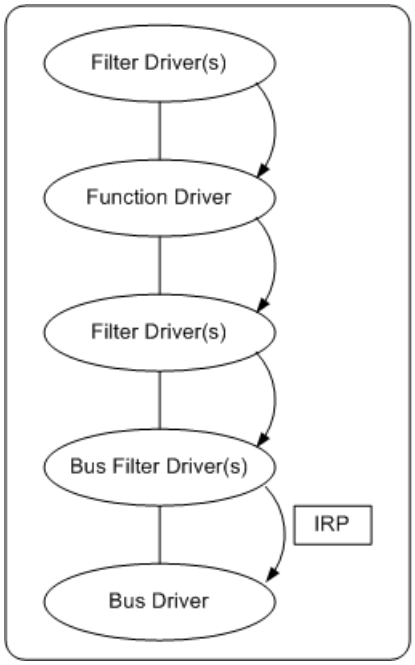
Passing PnP IRPs Down the Device Stack

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager uses IRPs to direct drivers to start, stop, and remove devices and to query drivers about their devices. All PnP IRPs have the major function code **IRP_MJ_PNP**, and all PnP drivers must provide a *DispatchPnP* routine to service this function code. The PnP manager initializes **Irp->IoStatus.Status** to STATUS_NOT_SUPPORTED when it sends an IRP. For more information, see DispatchPnP Routines.

For a list of PnP minor IRPs, see Plug and Play Minor IRPs.

All drivers for a device must have the opportunity to respond to a PnP IRP unless a driver in the stack fails the IRP. (See the following figure.)



No single driver for a device can assume that it is the only driver that will respond to a PnP IRP. Consider, for example, a function driver that responds to an **IRP_MN_QUERY_CAPABILITIES** request and completes the IRP without passing it to the next-lower driver. None of the capabilities supported by lower drivers, such as a unique instance ID or power management capabilities supported by the parent bus driver, is reported.

A PnP IRP travels back up the device stack when the parent bus driver calls **IoCompleteRequest** and the I/O manager calls any *IoCompletion* routines registered by the function driver or filter drivers.

A function or filter driver must do the following when it receives a PnP IRP:

- If the driver performs actions in response to the IRP:
  1. Perform the appropriate actions.
  2. Set **Irp->IoStatus.Status** to an appropriate status, such as STATUS_SUCCESS. Set **Irp->IoStatus.Information**, if appropriate for the IRP.
  3. Set up the next stack location with **IoSkipCurrentIrpStackLocation** or **IoCopyCurrentIrpStackLocationToNext**. Call the latter routine if you set an *IoCompletion* routine.
  4. Set an *IoCompletion* routine, if necessary.
  5. Do not complete the IRP. (Do not call **IoCompleteRequest**.) The parent bus driver will complete the IRP.
- If the driver does not perform actions for this IRP, it simply prepares to pass the IRP to the next driver:

1. Call **IoSkipCurrentIrpStackLocation** to remove its stack location from the IRP.
2. Do not set any fields in **Irp->IoStatus**.
3. Do not set an *IoCompletion* routine.
4. Do not complete the IRP. (Do not call **IoCompleteRequest**.) The parent bus driver will complete the IRP.

If a function or filter driver did not fail the IRP, it passes the IRP to the next-lower driver with **IoCallDriver**. A driver has a pointer to the next-lower driver; that pointer was returned from the **IoAttachDeviceToDeviceStack** call in the higher driver's *AddDevice* routine.

The parent bus driver completes the IRP after performing any tasks to respond to the IRP. After the bus driver calls **IoCompleteRequest**, the I/O manager calls any *IoCompletion* routines registered by the function or filter drivers for the device.

Postponing PnP IRP Processing Until Lower Drivers Finish

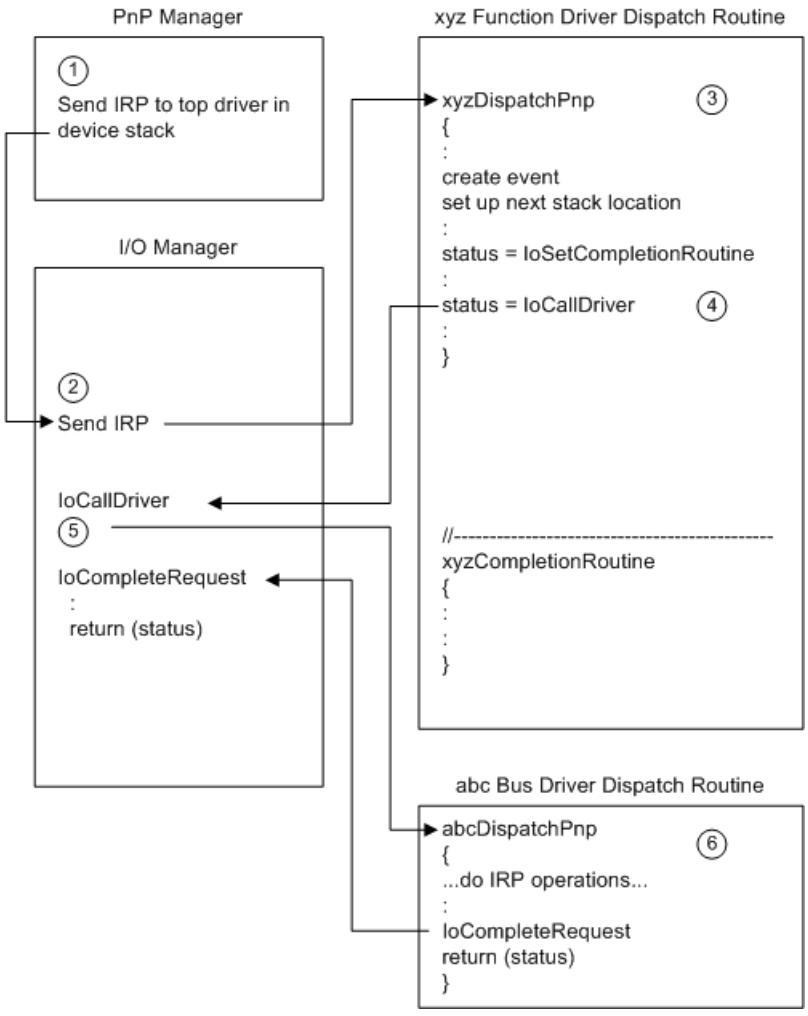
6/25/2019 • 5 minutes to read • Edit Online

Some PnP and power IRPs must be processed first by the parent bus driver for a device and then by each next-higher driver in the device stack. For example, the parent bus driver must be the first driver to perform its start operations for a device (**IRP_MN_START_DEVICE**), followed by each next-higher driver. For such an IRP, function and filter drivers must set an I/O completion routine, pass the IRP to the next-lower driver, and postpone any activities to process the IRP until the lower drivers have finished with the IRP.

An *IoCompletion* routine can be called at IRQL DISPATCH_LEVEL, but a function or filter driver might need to process the IRP at IRQL = PASSIVE_LEVEL. To return to PASSIVE_LEVEL from an *IoCompletion* routine, a driver can use a kernel event. The driver registers an *IoCompletion* routine that sets a kernel-mode event and then the driver waits on the event in its *DispatchPnP* routine. When the event is set, lower drivers have completed the IRP and the driver is allowed to process the IRP.

Note that a driver must not use this technique to wait for lower drivers to finish a power IRP (**IRP_MJ_POWER**). Waiting on an event in the *DispatchPower* routine that is set in the *IoCompletion* routine can cause a deadlock. See Passing Power IRPs for more information.

The following two figures show an example of how a driver waits for lower drivers to complete a PnP IRP. The example shows what the function and bus drivers must do, plus how they interact with the PnP manager and the I/O manager.



The following notes correspond to the circled numbers in the previous figure:

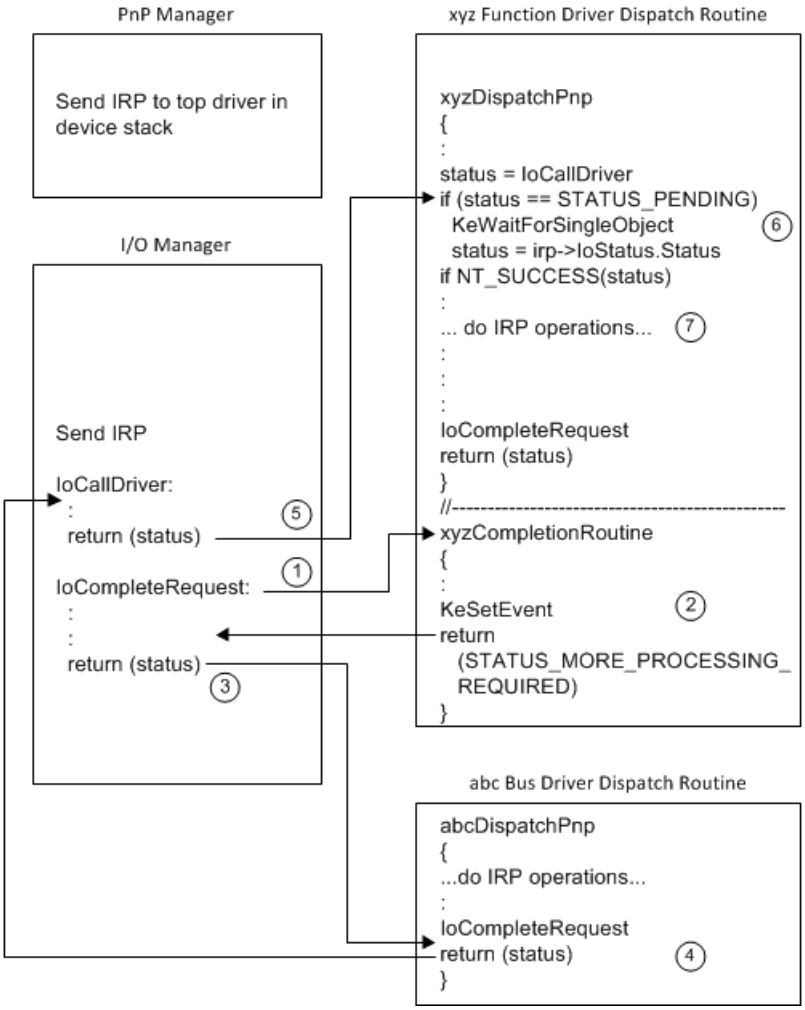
1. The PnP manager calls the I/O manager to send an IRP to the top driver in the device stack.
2. The I/O manager calls the *DispatchPnP* routine of the top driver. In this example, there are only two drivers in the device stack (the function driver and the parent bus driver) and the function driver is the top driver.
3. The function driver declares and initializes a kernel-mode event, sets up the stack location for the next-lower driver, and sets an *IoCompletion* routine for this IRP.

   The function driver can use **IoCopyCurrentIrpStackLocationToNext** to set up the stack location.

   In the call to **IoSetCompletionRoutine**, the function driver sets *InvokeOnSuccess*, *InvokeOnError*, and *InvokeOnCancel* to **TRUE** and passes the kernel-mode event as part of the context parameter.
4. The function driver passes the IRP down the device stack with **IoCallDriver** before performing any operations to handle the IRP.
5. The I/O manager sends the IRP to the next-lower driver in the device stack by calling that driver's *DispatchPnP* routine.
6. The next-lower driver in this example is the lowest driver in the device stack, the parent bus driver. The bus driver performs its operations to start the device. The bus driver sets **Irp->IoStatus.Status**, sets **Irp->IoStatus.Information** if relevant to this IRP, and completes the IRP by calling **IoCompleteRequest**.

   If the bus driver calls other driver routines or sends I/O to the device in order to start it, the bus driver does not complete the PnP IRP in its *DispatchPnP* routine. Instead, it must mark the IRP pending with **IoMarkIrpPending** and return STATUS_PENDING from its *DispatchPnP* routine. The driver later calls **IoCompleteRequest** from another driver routine, possibly a DPC routine.

The following figure shows the second part of the example, where the higher drivers in the device stack resume their postponed IRP processing.



The following notes correspond to the circled numbers in the previous figure:

1. When the bus driver calls **IoCompleteRequest**, the I/O manager examines the stack locations of the higher drivers and calls any *IoCompletion* routines it finds. In this example, the I/O manager locates and calls the *IoCompletion* routine for the next-higher driver, the function driver.

2. The function driver's *IoCompletion* routine sets the kernel-mode event supplied in the context parameter and returns STATUS_MORE_PROCESSING_REQUIRED.

   The IoCompletion routine must return STATUS_MORE_PROCESSING_REQUIRED to prevent the I/O manager from calling *IoCompletion* routines set by higher drivers at this time. The *IoCompletion* routine uses this status to forestall completion so its driver's *DispatchPnP* routine can regain control. The I/O manager will resume calling higher drivers' *IoCompletion* routines for this IRP when this driver's *DispatchPnP* routine completes the IRP.

3. The I/O manager stops completing the IRP and returns control to the routine that called **IoCompleteRequest**, which in this example is the bus driver's *DispatchPnP* routine.

4. The bus driver returns from its *DispatchPnP* routine with status indicating the result of its IRP processing: either STATUS_SUCCESS or an error status.

5. **IoCallDriver** returns control to its caller, which in this example is the function driver's *DispatchPnP* routine.

6. The function driver's *DispatchPnP* routine resumes processing the IRP.

   If **IoCallDriver** returns STATUS_PENDING, the *DispatchPnP* routine has resumed execution before its *IoCompletion* routine has been called. The *DispatchPnP* routine, therefore, must wait for the kernel event to be signaled by its *IoCompletion* routine. This ensures that the *DispatchPnP* routine will not continue processing the IRP until all lower drivers have completed it.

If **Irp->IoStatus.Status** is set to an error, a lower driver has failed the IRP and the function driver must not continue handling the IRP (except for any necessary cleanup).

7. Once lower drivers have successfully completed the IRP, the function driver processes the IRP.

   For IRPs being handled first by the parent bus driver, the bus driver typically sets a successful status in **Irp->IoStatus.Status** and optionally sets a value in **Irp->IoStatus.Information**. Function and filter drivers leave the values in **IoStatus** as is unless they fail the IRP.

   The function driver's *DispatchPnP* routine calls **IoCompleteRequest** to complete the IRP. The I/O manager resumes I/O completion processing. In this example, there are no filter drivers above the function driver, and thus no more *IoCompletion* routines to call. When **IoCompleteRequest** returns control to the function driver *DispatchPnP* routine, the *DispatchPnP* routine returns status.

For some IRPs, if a function or filter driver fails the IRP on its way back up the device stack, the PnP manager informs the lower drivers. For example, if a function or filter driver fails an **IRP_MN_START_DEVICE**, the PnP manager sends an **IRP_MN_REMOVE_DEVICE** to the device stack.

Starting a Device

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager sends an **IRP_MN_START_DEVICE** request to drivers either to start a newly enumerated device or to restart an existing device that was stopped for resource rebalancing.

Function and filter drivers must set an *IoCompletion* routine, pass the **IRP_MN_START_DEVICE** request down the device stack, and postpone their start operations until all lower drivers have finished with the IRP. The parent bus driver, the bottom driver in the device stack, must be the first driver to perform its start operations on a device before the device is accessed by other drivers.

To ensure proper sequencing of start operations, the PnP manager on Windows 2000 and later versions of Windows postpones exposing device interfaces and blocks create requests for the device until the start IRP succeeds.

If a driver for a device fails the **IRP_MN_START_DEVICE** request, the PnP manager sends an **IRP_MN_REMOVE_DEVICE** request to the device stack (on Windows 2000 and later versions of Windows). In response to this IRP, the drivers for the device undo their start operations (if they succeeded the start IRP), undo their *AddDevice* operations, and detach from the device stack. The PnP manager marks such a device "failed start."

This section covers the following topics:

Starting a Device in a Function Driver

6/25/2019 • 4 minutes to read • Edit Online

A function driver sets an *IoCompletion* routine, passes an **IRP_MN_START_DEVICE** request down the device stack, and postpones its start operations until all lower drivers have finished with the IRP. See Postponing PnP IRP Processing Until Lower Drivers Finish for detailed information about using a kernel event and an *IoCompletion* routine to postpone IRP processing.

When its *DispatchPnP* routine regains control after all lower drivers have finished with the IRP, the function driver performs its tasks for starting the device. A function driver starts the device with a procedure like the following:

1. If a lower driver failed the IRP (**IoCallDriver** returned an error), do not continue processing the IRP. Do any necessary cleanup and return from the *DispatchPnP* routine (go to the last step in this list).

2. If lower drivers processed the IRP successfully, start the device.

   The exact steps to start a device vary from device to device. Such steps might include mapping I/O space, initializing hardware registers, setting the device in the D0 power state, and connecting the interrupt with **IoConnectInterrupt**. If the driver is restarting a device after an **IRP_MN_STOP_DEVICE** request, the driver might have device state to restore.

   The device must be powered on before any drivers can access it. See Powering Up a Device for more information.

   If the device should be enabled for wake-up, its power policy owner (usually the function driver) should send a wait/wake IRP after it powers up the device and before it completes the **IRP_MN_START_DEVICE** request. For details, see Sending a Wait/Wake IRP.

3. Start IRPs in the IRP-holding queue.

   Clear the driver-defined HOLD_NEW_REQUESTS flag and start the IRPs in the IRP-holding queue. Drivers should do this when starting a device for the first time and when restarting a device after a query-stop or stop IRP. See Holding Incoming IRPs When A Device Is Paused for more information.

4. [Optional] Enable interfaces for the device by calling **IoSetDeviceInterfaceState**.

   Enable the interfaces, if any, that the driver previously registered in its *AddDevice* routine (or in an INF or by another component such as a co-installer).

   On Windows 2000 and later versions of Windows, the PnP manager does not send notification of device-interface arrivals until the **IRP_MN_START_DEVICE** IRP completes, indicating that all the drivers for the device have completed their start operations. The PnP manager also fails any create requests that arrive before all the drivers for the device complete the start IRP.

5. Complete the IRP.

   The function driver's *IoCompletion* routine returned STATUS_MORE_PROCESSING_REQUIRED, as described in Postponing PnP IRP Processing Until Lower Drivers Finish, so the function driver's *DispatchPnP* routine must call **IoCompleteRequest** to resume I/O completion processing.

   If the function driver's start operations were successful, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS, calls **IoCompleteRequest** with a priority boost of IO_NO_INCREMENT, and returns STATUS_SUCCESS from its *DispatchPnP* routine.

   If the function driver encounters an error during its start operations, the driver sets an error status in the

IRP, calls **IoCompleteRequest** with IO_NO_INCREMENT, and returns the error from its *DispatchPnP* routine.

If a lower driver failed the IRP (**IoCallDriver** returned an error), the function driver calls **IoCompleteRequest** with IO_NO_INCREMENT and returns the **IoCallDriver** error from its *DispatchPnP* routine. The function driver does not set **Irp->IoStatus.Status** in this case because the status has already been set by the lower driver that failed the IRP.

When a function driver receives an **IRP_MN_START_DEVICE** request, it should examine the structures at **IrpSp->Parameters.StartDevice.AllocatedResources** and **IrpSp->Parameters.StartDevice.AllocatedResourcesTranslated**, which describe the raw and translated resources, respectively, that the PnP manager has assigned to the device. Drivers should save a copy of each resource list in the device extension as a debugging aid.

The resource lists are paired **CM_RESOURCE_LIST** structures, in which each element of the raw list corresponds to the same element of the translated list. For example, if **AllocatedResources.List**[0] describes a raw I/O port range, then **AllocatedResourcesTranslated.List**[0] describes the same range after translation. Each translated resource includes a physical address and the type of the resource.

If a driver is assigned a translated memory resource (**CmResourceTypeMemory**), it must call **MmMapIoSpace** to map the physical address into a virtual address through which it can access device registers. For a driver to operate in a platform independent manner, it should check each returned, translated resource and map it, if necessary.

A function driver should do the following in response to an **IRP_MN_START_DEVICE** to ensure access to all device resources:

1. Copy **IrpSp->Parameters.StartDevice.AllocatedResources** to the device extension.
2. Copy **IrpSp->Parameters.StartDevice.AllocatedResourcesTranslated** to the device extension.
3. In a loop, inspect each descriptor element in **AllocatedResourcesTranslated**. If the descriptor resource type is **CmResourceTypeMemory**, call **MmMapIoSpace**, passing the physical address and length of the translated resource.

When the driver receives an **IRP_MN_STOP_DEVICE**, **IRP_MN_REMOVE_DEVICE**, or **IRP_MN_SURPRISE_REMOVAL** request, it must release the mappings by calling **MmUnmapIoSpace** in a similar loop. The driver should also call **MmUnmapIoSpace** if it must fail the **IRP_MN_START_DEVICE** request.

See Mapping Bus-Relative Addresses to Virtual Addresses for more information.

Starting a Device in a Filter Driver

12/5/2018 • 2 minutes to read • Edit Online

An upper-level filter driver might augment any of the start activities of the function driver.

A lower-level filter typically augments the features of the device and might participate in starting the device.

Starting a Device in a Bus Driver

6/25/2019 • 2 minutes to read • Edit Online

A bus driver starts a child device (child *PDO*) with a procedure such as the following in its *DispatchPnP* routine:

1. Start the device.

   The exact steps vary from device to device.

   For example, the PCI bus driver programs its mapping registers to enable requests on the PCI bus. The PnP ISA bus driver enables the PnP ISA card so the function driver can access it.

2. Complete the IRP.

   If the bus driver's start operations were successful, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS and calls **IoCompleteRequest** specifying a priority boost of IO_NO_INCREMENT. The bus driver returns STATUS_SUCCESS from its *DispatchPnP* routine.

   If the bus driver encounters an error during its start operations, the driver sets an error status in the IRP, calls **IoCompleteRequest** with IO_NO_INCREMENT, and returns the error from its *DispatchPnP* routine.

If a bus driver requires some time to start the device, it can mark the IRP as pending and return STATUS_PENDING.

Design Guidelines for Starting Devices

6/25/2019 • 2 minutes to read • Edit Online

- The PnP manager fails create requests for the device until the **IRP_MN_START_DEVICE** IRP completes, indicating that all the drivers for the device have performed their start operations.
- Because a *DispatchPnP* routine runs in the context of a system thread at IRQL PASSIVE_LEVEL, any memory allocated with **ExAllocatePoolWithTag** for use exclusively during initialization can be from paged pool as long as the driver does not control the device that holds a system page file. Such a memory allocation must be released with **ExFreePool** before the *DispatchPnP* routine returns control.
- A WDM device driver's ISR should be capable of determining whether it has been called with a spurious interrupt even during device startup. On return from the call to **IoConnectInterrupt** in the code path that handles **IRP_MN_START_DEVICE**, the ISR can be called immediately if interrupts are enabled on the device.

Stopping a Device

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager directs drivers to stop a device in the following situations:

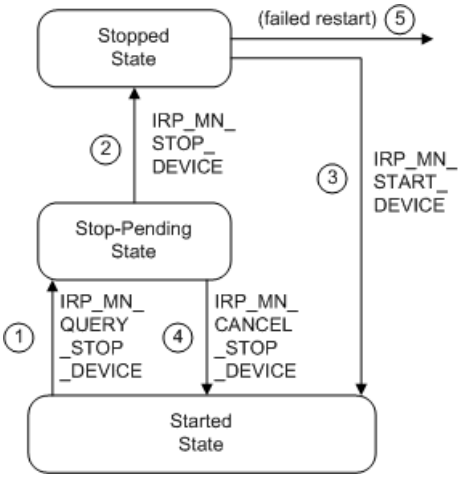
- To rebalance the hardware resources being used by the device. Rebalancing is typically necessary when a new device is enumerated that requires a resource already in use.
- To disable the device in response to a Device Manager request (Windows 98/Me only). Windows 2000 and later versions of Windows send remove IRPs in this situation; see Understanding When Remove IRPs Are Issued.
- After a failed **IRP_MN_START_DEVICE** request (Windows 98/Me only)

This section covers the following topics:

Stopping a Device to Rebalance Resources

6/25/2019 • 2 minutes to read • Edit Online

The following figure shows the sequence of IRPs involved in stopping and restarting a device to rebalance resources.



The following notes correspond to the circled numbers in the previous figure:

1. The PnP manager issues an **IRP_MN_QUERY_STOP_DEVICE** to ask whether the drivers for a device can stop the device and release its hardware resources.

   If all the drivers in the device stack return STATUS_SUCCESS, the drivers have put the device into a state (stop-pending) from which the device can be quickly stopped.

   The PnP manager queries as many device stacks as necessary to rebalance the required resources.

2. The PnP manager issues an **IRP_MN_STOP_DEVICE** to stop the device.

   On Windows 2000 and later versions of Windows, the PnP manager sends a stop IRP only if a previous query-stop IRP for the device completed successfully. In response to a stop IRP, drivers release the device's hardware resources (such as its I/O ports) and hold any IRPs that require access to the device.

3. After successfully rebalancing resources, the PnP manager issues **IRP_MN_START_DEVICE** requests to restart any devices that it stopped during the rebalance.

4. Otherwise, the PnP manager cancels a query-stop IRP by sending an **IRP_MN_CANCEL_STOP_DEVICE**.

   In response to an **IRP_MN_CANCEL_STOP_DEVICE**, the drivers for a device return the device to the started state and resume processing I/O requests for the device.

   The PnP manager cancels the query-stop for a device stack if one driver in the stack failed the request or if the overall rebalance operation failed and it is canceling all its query-stop requests. When the PnP manager cancels the query-stop on just one device stack, it sends the **IRP_MN_CANCEL_STOP_DEVICE** request because any drivers attached above the driver that failed the query have the device in the stop-pending state. When the **IRP_MN_CANCEL_STOP_DEVICE** succeeds, drivers have returned the device to the started state.

5. If a driver fails to restart the device after rebalancing resources, the PnP manager sends remove IRPs to the device stack (on Windows 2000 and later versions of Windows).

   The PnP manager first sends an **IRP_MN_SURPRISE_REMOVAL** request. Then it sends an

IRP_MN_REMOVE_DEVICE request, but only after all open handles to the device are closed.

Rebalancing the hardware resources of a PnP device must be transparent to applications and end users. Users might experience a temporary delay in operation, but data must not be lost. You must take that into consideration when you handle stop IRPs.

Stopping a Device to Disable It (Windows 98/Me)

6/25/2019 • 2 minutes to read • Edit Online

On Windows 98/Me, the PnP manager issues stop IRPs when Device Manager disables the device. (Windows 2000 and later versions of Windows issue remove IRPs in this situation).

The PnP manager sends the stop IRPs in the following sequence:

1. The PnP manager issues an **IRP_MN_QUERY_STOP_DEVICE** to ask whether the drivers for a device can stop the device.

   If all the drivers in the device stack return STATUS_SUCCESS, the drivers have put the device into a state (stop-pending) from which the device can be quickly stopped.

   The PnP manager queries as many device stacks as necessary to disable the device.

2. If the **IRP_MN_QUERY_STOP_DEVICE** succeeds, the PnP manager issues an **IRP_MN_STOP_DEVICE** to stop the device.

   The PnP manager sends the stop IRP only if the previous query-stop IRP for the device completed successfully. In response to the stop IRP, drivers release the device's hardware resources (such as its I/O ports) and fail any IRPs that require access to the device.

3. If the **IRP_MN_QUERY_STOP_DEVICE** fails, the PnP manager sends an **IRP_MN_CANCEL_STOP_DEVICE** to cancel the query.

   In response to an **IRP_MN_CANCEL_STOP_DEVICE**, the drivers for a device return the device to the started state and resume processing I/O requests for the device.

   The PnP manager cancels the query-stop for a device stack if one driver in the stack failed the request. When the PnP manager cancels the query-stop on just one device stack, it sends the **IRP_MN_CANCEL_STOP_DEVICE** request because any drivers attached above the driver that failed the query have the device in the stop-pending state. When the **IRP_MN_CANCEL_STOP_DEVICE** succeeds, drivers have returned the device to the started state.

When a device is being disabled, its drivers cannot queue incoming IRPs because there is no guarantee when the device might be reenabled. Consequently, data might be lost.

Stopping a Device after a Failed Start (Windows 98/Me)

6/25/2019 • 2 minutes to read • Edit Online

On Windows 98/Me, the PnP manager issues an **IRP_MN_STOP_DEVICE** request without a preceding query when the drivers for a device fail an **IRP_MN_START_DEVICE** request. (On Windows 2000 and later, the PnP manager sends remove IRPs in this situation. See Understanding When Remove IRPs Are Issued.)

In response to the stop IRP, drivers release the device's hardware resources (such as its I/O ports), disable and deregister any user-mode interfaces, and fail any incoming I/O requests that require access to the device.

Handling Stop IRPs (Windows 2000 and Later)

12/5/2018 • 2 minutes to read • Edit Online

Drivers that run only on Windows 2000 and later versions of Windows (WDM versions 0x10 and greater) receive stop IRPs only when the PnP manager rebalances resources. The following sections describe techniques such drivers should use in handling stop IRPs:

Handling an IRP_MN_QUERY_STOP_DEVICE Request (Windows 2000 and later)

Handling an IRP_MN_STOP_DEVICE Request (Windows 2000 and later)

Handling an IRP_MN_CANCEL_STOP_DEVICE Request (Windows 2000 and later)

Holding Incoming IRPs When A Device Is Paused

WDM drivers that also run on Windows 98/Me must handle these IRPs differently. See Handling Stop IRPs (Windows 98/Me) for details.

Handling an IRP_MN_QUERY_STOP_DEVICE Request (Windows 2000 and later)

6/25/2019 • 3 minutes to read • Edit Online

An **IRP_MN_QUERY_STOP_DEVICE** request is handled first by the top driver in the device stack and then by each next lower driver. A driver handles stop IRPs in its *DispatchPnP* routine.

In response to an **IRP_MN_QUERY_STOP_DEVICE**, a driver must do the following:

1. Determine whether the device can be stopped, and its hardware resources released, without adverse affects.

   A driver must fail a query-stop IRP if any of the following are true:

   - A driver has been notified (through **IRP_MN_DEVICE_USAGE_NOTIFICATION**) that the device is in the path of a paging, hibernation, or crash dump file.
   - The device's hardware resources cannot be released.

   A driver might fail a query-stop IRP if the following is true:

   - The driver must not drop I/O requests and does not have a mechanism for queuing IRPs.

     While the device is in the stopped state, a driver must hold IRPs that require access to the device. If a driver does not queue IRPs, it must not allow the device to be stopped and thus must fail a query-stop IRP.

     The exception to this rule is a device that is allowed to drop I/O. The drivers for such a device can succeed query-stop and stop requests without queuing IRPs.

2. If the device cannot be stopped, fail the query-stop IRP.

   Set **Irp->IoStatus.Status** to an appropriate error status, call **IoCompleteRequest** with IO_NO_INCREMENT, and return from the driver's *DispatchPnP* routine. Do not pass the IRP to the next lower driver.

3. If the device can be stopped and the driver queues IRPs, set the HOLD_NEW_REQUESTS flag in the device extension so subsequent IRPs will be queued (see Holding Incoming IRPs When A Device Is Paused).

   Alternatively, the drivers for a device can defer completely pausing the device until the drivers receive the subsequent **IRP_MN_STOP_DEVICE** request. Such drivers, however, must queue any requests that would prevent them from immediately succeeding the stop IRP when it arrives. Until the device is restarted, such drivers must queue requests such as the following:

   - **IRP_MN_DEVICE_USAGE_NOTIFICATION** requests (for example, to put a paging file on the device).
   - Requests for isochronous transfers.
   - Create requests that would prevent the drivers from succeeding a stop IRP.

4. If the device cannot have an IRP in progress fail, ensure that any outstanding requests that were passed to other driver routines and to lower drivers have completed.

   One way that a driver can achieve this is to use a reference count and an event to ensure that all requests have been completed:

- In its *AddDevice* routine, the driver defines an I/O reference count in the device extension and initializes the count to one.
- Also in its *AddDevice* routine, the driver creates an event with **KeInitializeEvent** and initializes the event to the Not-Signaled state with **KeClearEvent**.
- Each time it processes an IRP, the driver increments the reference count with **InterlockedIncrement**.
- Each time it completes a request, the driver decrements the reference count with **InterlockedDecrement**.

  The driver decrements the reference count in the *IoCompletion* routine, if the request has one, or right after the call to **IoCallDriver** if the driver uses no *IoCompletion* routine for the request.
- When the driver receives an **IRP_MN_QUERY_STOP_DEVICE**, it decrements the reference count with **InterlockedDecrement**. If there are no outstanding requests, this reduces the reference count to zero.
- When the reference count reaches zero, the driver sets the event with **KeSetEvent** signaling that the query-stop code can continue.

As an alternative to the above procedure, a driver can serialize the **IRP_MN_QUERY_STOP_DEVICE** IRP behind any IRPs in progress.

5. Perform any other steps required to put the device in the stop-pending state.

   After a driver succeeds a query-stop IRP, it must be ready to succeed an **IRP_MN_STOP_DEVICE**.

6. Finish the IRP.

   In a function or filter driver:

   - Set **Irp->IoStatus.Status** to STATUS_SUCCESS.
   - Set up the next stack location with **IoSkipCurrentIrpStackLocation** and pass the IRP to the next lower driver with **IoCallDriver**.
   - Propagate the status from **IoCallDriver** as the return status from the *DispatchPnP* routine.
   - Do not complete the IRP.

   In a bus driver:

   - Set **Irp->IoStatus.Status** to STATUS_SUCCESS.

     If, however, the devices on the bus use hardware resources, reevaluate the resource requirements of the bus and the child devices. If any of the requirements have changed, return STATUS_RESOURCE_REQUIREMENTS_CHANGED instead of STATUS_SUCCESS. This status indicates success but requests that the PnP manager requery your resources before sending the stop IRP.
   - Complete the IRP (**IoCompleteRequest**) with IO_NO_INCREMENT.
   - Return from the *DispatchPnP* routine.

If any driver in the device stack fails the **IRP_MN_QUERY_STOP_DEVICE**, the PnP manager sends an **IRP_MN_CANCEL_STOP_DEVICE** to the device stack. This prevents drivers from requiring an *IoCompletion* routine for a query-stop IRP to detect whether a lower driver failed the IRP.

Handling an IRP_MN_STOP_DEVICE Request (Windows 2000 and later)

6/25/2019 • 2 minutes to read • Edit Online

An **IRP_MN_STOP_DEVICE** request is handled first by the top driver in the device stack and then by each next lower driver. A driver handles stop IRPs in its *DispatchPnP* routine.

A driver handles an **IRP_MN_STOP_DEVICE** request with a procedure such as the following:

1. Ensure that the device is paused.

   If a driver did not completely pause the device in response to the **IRP_MN_QUERY_STOP_DEVICE** request, it must do so now. Set a HOLD_NEW_REQUESTS flag in the device extension and perform any other necessary operations to pause the device.

   The device might lose power during the resource-rebalance operation and thus might lose device state. Drivers for the device should save any device state information and restore it when they receive the subsequent **IRP_MN_START_DEVICE** request.

2. Release the hardware resources for the device.

   In a function driver, the exact operations depend on the device and the driver but can include disconnecting an interrupt with **IoDisconnectInterrupt**, freeing physical address ranges with **MmUnmapIoSpace**, and freeing I/O ports.

   If a filter or bus driver acquired any hardware resources for the device, that driver must release the resources in response to an **IRP_MN_STOP_DEVICE** request.

3. Set **Irp->IoStatus.Status** to STATUS_SUCCESS.

4. Pass the IRP to the next lower driver or complete the IRP.

   - In a function or filter driver, set up the next stack location with **IoSkipCurrentIrpStackLocation**, pass the IRP to the next lower driver with **IoCallDriver**, and return the status from **IoCallDriver** as the return status from the *DispatchPnP* routine. Do not complete the IRP.
   - In a bus driver, complete the IRP using **IoCompleteRequest** with IO_NO_INCREMENT and return from the *DispatchPnP* routine.

While the device is stopped to rebalance resources, a driver cannot start any IRPs that access the device. A driver must queue such IRPs, as described in Holding Incoming IRPs When A Device Is Paused, or fail them if the driver does not implement an IRP-holding queue and must not drop I/O requests.

Handling an IRP_MN_CANCEL_STOP_DEVICE Request (Windows 2000 and later)

6/25/2019 • 2 minutes to read • Edit Online

An **IRP_MN_CANCEL_STOP_DEVICE** request must be handled first by the parent bus driver for a device and then by each next higher driver in the device stack. A driver handles stop IRPs in its *DispatchPnP* routine.

In response to an **IRP_MN_CANCEL_STOP_DEVICE** request, a driver must return the device to its started state and resume normal operation. Drivers must succeed a cancel-stop IRP.

A driver handles an **IRP_MN_CANCEL_STOP_DEVICE** request with a procedure such as the following:

1. Postpone restarting the device until lower drivers have completed their restart operations. (See Postponing PnP IRP Processing Until Lower Drivers Finish.)
2. After lower drivers finish, return the device to its started state.

   Exact operations depend on the device and the driver.
3. Start IRPs in the IRP-holding queue.

   If the driver was holding requests while the device was in the stop-pending state, clear the HOLD_NEW_REQUESTS flag and start the IRPs in the IRP-holding queue. See Holding Incoming IRPs When A Device Is Paused for more information.
4. Complete the IRP with **IoCompleteRequest**.

   - In a function or filter driver:

     The driver's *IoCompletion* routine returned STATUS_MORE_PROCESSING_REQUIRED, as described in Postponing PnP IRP Processing Until Lower Drivers Finish, so the driver's *DispatchPnP* routine must call **IoCompleteRequest** to resume I/O completion processing.

     The driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS, calls **IoCompleteRequest** with a priority boost of IO_NO_INCREMENT, and returns STATUS_SUCCESS from its *DispatchPnP* routine.

     Drivers must not fail this operation. If a driver fails the restart IRP, the device is in an inconsistent state and will not operate properly.

   - In a parent bus driver:

     The driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS and calls **IoCompleteRequest** specifying a priority boost of IO_NO_INCREMENT. The bus driver returns STATUS_SUCCESS from its *DispatchPnP* routine.

     A bus driver must not fail this operation. If a driver fails the restart IRP, the device is in an inconsistent state and will not operate properly.

A driver might receive a spurious cancel-stop request when the device is started and active. This can occur, for example, if the driver (or a driver higher in the device stack) failed an **IRP_MN_QUERY_STOP_DEVICE** request. When a device is started and active, drivers can safely succeed spurious cancel-stop requests for the device.

Holding Incoming IRPs When A Device Is Paused

6/25/2019 • 2 minutes to read • Edit Online

The drivers for a device must pause the device when its resources are being rebalanced. During resource rebalancing, some drivers pause the device in response to an **IRP_MN_QUERY_STOP_DEVICE** request and other drivers delay pausing the device until they receive the **IRP_MN_STOP_DEVICE** request. In either case, the device must be paused when the **IRP_MN_STOP_DEVICE** succeeds.

The drivers must finish any IRPs in progress on the device and refrain from starting any new IRPs that require access to the device.

To hold IRPs while a device is paused, a driver implements a procedure such as the following:

1. In its *AddDevice* routine, define a flag in the device extension with a name like HOLD_NEW_REQUESTS. Clear the flag.

2. Create a FIFO queue for holding IRPs.

   If the driver already queues IRPs, it can reuse the same queue because the driver is required to finish any outstanding requests before pausing the device.

   If the driver does not already have an IRP queue, it must create one in its *AddDevice* routine. What kind of queue it creates depends on how the driver flushes the queue. A driver might use an interlocked, doubly linked list and the **ExInterlocked*Xxx*List** routines.

3. In its *DispatchPnP* code for **IRP_MN_QUERY_STOP_DEVICE** (or **IRP_MN_STOP_DEVICE**), finish any outstanding requests and set the HOLD_NEW_REQUESTS flag.

4. In a dispatch routine that accesses the device, such as *DispatchWrite* or *DispatchRead*, check whether the HOLD_NEW_REQUESTS flag is set. If so, the driver must mark the IRP pending and queue it.

   The driver's *DispatchPnP* routine must continue to process PnP IRPs rather than hold them and the *DispatchPower* routine must continue to process power IRPs.

5. In *DispatchPnP*, in response to a start or cancel-stop IRP, clear the HOLD_NEW_REQUESTS flag and start the IRPs in the IRP-holding queue.

   These actions are probably the last steps for processing these PnP IRPs. For example, in response to a start IRP, the driver must first perform any operations to start the device and then it can start the IRPs in the IRP-holding queue.

   Errors in processing IRPs from the IRP-holding queue do not affect the status returned for the start or cancel-stop IRPs.

Resetting and recovering a device

12/5/2018 • 6 minutes to read • Edit Online

The GUID_DEVICE_RESET_INTERFACE_STANDARD interface defines a standard way for function drivers to attempt to reset and recover a malfunctioning device.

Two types of device resets are available through this interface:

- Function-level device reset. In this case, the reset operation is restricted to a specific device, and is not visible to other devices. The device stays connected to the bus throughout the reset and returns to a valid state (initial state) after the reset. This type of reset has the least impact on the system.

  This type of reset can be implemented either by the bus driver or by ACPI firmware. The bus driver can implement a function-level reset if the bus specification defines an in-band reset mechanism that meets the requirement. ACPI firmware can optionally override a bus driver-defined function-level reset with its own implementation.

- Platform-level device reset. In this case, the reset operation causes the device to be reported as missing from the bus. The reset operation affects a specific device and all other devices that are connected to it via the same power rail or reset line. This type of reset has the most impact on the system. The OS will tear down and rebuild the stacks of all affected devices to ensure that everything restarts from a blank state.

  Starting in Windows 10, these registry entries under the HKLM\SYSTEM\CurrentControlSet\Control\Pnp key configures the reset operation:

  - DeviceResetRetryInterval: Time period before the reset operation starts. Default value is 3 seconds. Minimum value is 100 milliseconds; maximum value is 30 seconds.
  - DeviceResetMaximumRetries: Number of times the reset operation is attempted.

Note The GUID_DEVICE_RESET_INTERFACE_STANDARD interface is available starting in Windows 10.

Using the device reset interface

If a function driver detects that the device is not functioning correctly, it should first attempt a function-level reset. If a function-level reset does not fix the issue, then the driver may choose to attempt a platform-level reset. However, a platform-level reset should only be used as the final option.

To query for this interface, a device driver sends an IRP_MN_QUERY_INTERFACE IRP down the driver stack. For this IRP, the driver sets the InterfaceType input parameter to GUID_DEVICE_RESET_INTERFACE_STANDARD. On successful completion of the IRP, the Interface output parameter is a pointer to a DEVICE_RESET_INTERFACE_STANDARD structure. This structure contains a pointer to the DeviceReset routine, which can be used to request a function-level or platform-level reset.

Supporting the device reset interface in function drivers

To support the device reset interface, the device stack must meet the following requirements.

The function driver must properly handle IRP_MN_QUERY_REMOVE_DEVICE, IRP_MN_REMOVE_DEVICE and IRP_MN_SURPRISE_REMOVAL.

In most cases, when the driver receives IRP_MN_QUERY_REMOVE_DEVICE, it should return a success so that the device can be safely removed. However, there may be cases where the device cannot be safely stopped, such as if the device is stuck in a loop writing to a memory buffer. In such cases, the driver should return

STATUS_DEVICE_HUNG to IRP_MN_QUERY_REMOVE_DEVICE. The PnP manager will continue the IRP_MN_QUERY_REMOVE_DEVICE and IRP_MN_REMOVE_DEVICE process, but that particular stack will not receive IRP_MN_REMOVE_DEVICE. Instead, the device stack will receive IRP_MN_SURPRISE_REMOVAL after the device has been reset.

For more information about these IRPs, see:

Handling an IRP_MN_QUERY_REMOVE_DEVICE Request

Handling an IRP_MN_REMOVE_DEVICE Request

Handling an IRP_MN_SURPRISE_REMOVAL Request

Supporting the device reset interface in filter drivers

Filter drivers may intercept IRP_MN_QUERY_INTERFACE IRPs that have the GUID_DEVICE_RESET_INTERFACE_STANDARD interface type. By doing so, they can continue to delegate to the GUID_DEVICE_RESET_INTERFACE_STANDARD interface but perform device-specific operations before or after the reset operation. Alternatively, they can override the GUID_DEVICE_RESET_INTERFACE_STANDARD interface returned by the bus driver with its own interface in order to provide its own reset operation.

Supporting the device reset interface in bus drivers

Bus drivers that participate in the device reset process (that is, bus drivers that are associated with the device that is requesting the reset and bus drivers that are associated with devices that are responding to the reset request) must meet one of the following requirements:

- Be hot plug capable. The bus driver must be able to detect a device being removed from the bus without notice, and a device being plugged into the bus.
- Alternatively, it must implement the GUID_REENUMERATE_SELF_INTERFACE_STANDARD interface. This simulates a device being pulled from a bus and being plugged back in. Built-in bus drivers (such as PCI and SDBUS) support this interface. Therefore, if the device being reset uses one of these buses, no bus driver modifications should be necessary.

  For WDF-based bus drivers, the WDF framework registers the GUID_REENUMERATE_SELF_INTERFACE_STANDARD interface on behalf of the drivers. Therefore, registering this interface is not necessary for those drivers. If the bus driver needs to do perform some operations before its child devices are re-enumerated, it must register for the EvtChildListDeviceReenumerated callback routine and perform the operations in that routine. Because this callback routine may be called in parallel for all PDO's, the code in the routine may need to protect against race conditions.

ACPI firmware: Function-level reset

To support function-level device reset, there must be an _RST method defined inside the Device scope. If present, this method will override the bus driver's implementation of function-level device reset (if present) for that device. When executed, the _RST method must reset only that device, and must not affect other devices. In addition, the device must stay connected on the bus.

ACPI firmware: Platform-level reset

To support platform-level device reset, there are two options:

- The ACPI firmware can define a PowerResource that implements the _RST method, and all devices that are affected by this reset method can refer to this PowerResource through a _PRR object defined under their

Device scope.

- The device can declare a _PR3 object. In this case, the ACPI driver will use D3cold power cycling to perform the reset, and reset dependencies between devices will be determined from the _PR3 object.

If the _PRR object exists in the Device scope, the ACPI driver will use the _RST method in the referenced PowerResource to perform the reset. If no _PRR object is defined but the _PR3 object is defined, then the ACPI driver will use D3cold power cycling to perform the reset. If neither the _PRR or _PR3 object is defined, then the device does not support a platform-level reset and the ACPI driver will report that the platform-level reset is not available.

Verifying ACPI firmware on the test system

To test your driver that supports device reset and recovery, follow this procedure. This procedure assumes you are using this example ASL file.

```
DefinitionBlock("SSDT.AML", "SSDT", 0x01, "XyzOEM", "TestTabl", 0x00001000)
{
    Scope(\_SB_)
       {
        PowerResource(PWFR, 0x5, 0x0)
        {
            Method(_RST, 0x0, NotSerialized)    { }

            // Placeholder methods as power resources need _ON, _OFF, _STA.
            Method(_STA, 0x0, NotSerialized)
            {
                Return(0xF)
            }

            Method(_ON_, 0x0, NotSerialized)    { }

            Method(_OFF, 0x0, NotSerialized)    { }

        } // PowerResource()
    } // Scope (\_SB_)

    // Assumes WiFi device is declared under \_SB.XYZ.
    Scope(\_SB_.XYZ.WIFI)
        {

        // Declare PWFR as WiFi reset power rail
        Name(_PRR, Package(One)
            {
                \_SB_.PWFR
            })
        } // Scope (\_SB)
}
```

1. Compile the test ASL file to an AML by using an ASL compiler, such as Asl.exe. The executable in included in the Windows Driver Kit (WDK). Asl .asl

   The preceding command generates SSDT.aml.

2. Rename SSDT.aml to acpitabl.dat.

3. Copy acpitabl.dat to %systemroot%\system32 on the test system.

4. Enable test signing on the test system. Bcdedit /set GUID_DEVICE_RESET_INTERFACE_STANDARD testsigning on

5. Reboot the test system.

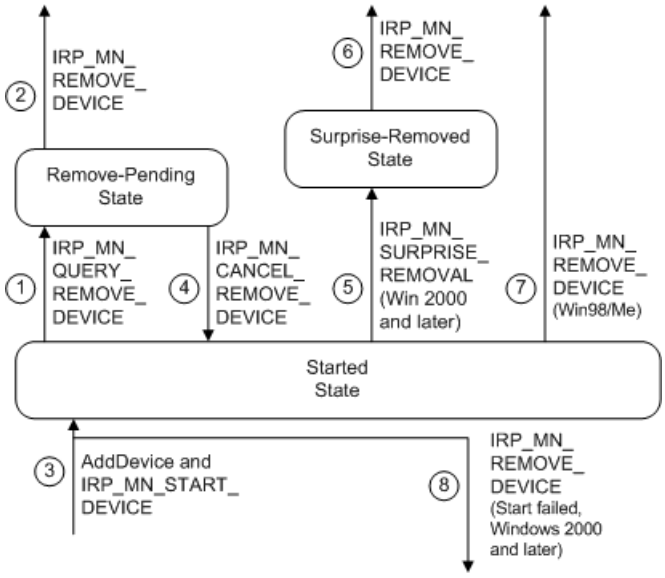
6. Verify that the table is loaded. In Windows Debugger, use these commands.

```
!acpicache
dt _DESCRIPTION_HEADER address of the SSDT table

0: kd> !acpicache
Dumping cached ACPI tables...
  SSDT @(ffffffffffd03018) Rev: 0x1 Len: 0x000043 TableID: TestTabl
  XSDT @(ffffffffffd05018) Rev: 0x1 Len: 0x000114 TableID: HSW-FFRD
       ...
       ...

0: kd> dt _DESCRIPTION_HEADER ffffffffffd03018
ACPI!_DESCRIPTION_HEADER
   +0x000 Signature        : 0x54445353
   +0x004 Length           : 0x43
   +0x008 Revision         : 0x1 ''
   +0x009 Checksum         : 0x37 '7'
   +0x00a OEMID            : [6]  "XyzOEM"
   +0x010 OEMTableID       : [8]  "TestTabl"
   +0x018 OEMRevision      : 0x1000
   +0x01c CreatorID        : [4]  "MSFT"
   +0x020 CreatorRev       : 0x5000000
```

Understanding When Remove IRPs Are Issued

6/25/2019 • 4 minutes to read • Edit Online

The following figure shows the typical sequence of IRPs involved in removing the drivers for a device.



The following notes correspond to the circled numbers in the previous figure:

1. Query remove

   The PnP manager issues an **IRP_MN_QUERY_REMOVE_DEVICE** to ask whether a device can be removed without disrupting the machine. It also sends this IRP when a user requests to update driver(s) for the device and (on Windows 2000 and later) when Device Manager disables the device. (On Windows 98/Me, the PnP manager sends stop IRPs in this situation; see Stopping a Device for details.)

   If all drivers in the device stack return STATUS_SUCCESS, the drivers have put the device into the remove-pending state. In this state, the drivers must not start any operations that prevent the device from being removed.

   In this "clean" removal case, the PnP manager sends a query-remove IRP before it sends a remove IRP. See step 5 for a description of "surprise" removal.

   Although it is not shown in the above diagram, a bus driver might receive an **IRP_MN_QUERY_REMOVE_DEVICE** for a device that is not started. This can happen if a user requests to dynamically remove a device that is physically present on the machine but is disabled.

2. Remove after successful query

   The PnP manager issues an **IRP_MN_REMOVE_DEVICE** to remove the drivers for a device.

   Drivers must succeed this request. The drivers for the device perform any necessary clean-up, detach from the device stack, and delete the FDO and any filter DOs. The parent bus driver retains the PDO until the user physically removes the device from the machine.

   Note that drivers might receive an **IRP_MN_STOP_DEVICE** prior to a remove IRP, but it is not required. On Windows 2000 and later, **IRP_MN_STOP_DEVICE** is used only to pause a device for resource rebalancing; it is not a step toward removal. If a user removes the device hardware while the device is stopped, the PnP manager sends a remove IRP at some point after the stop IRP, but a stop is not a prerequisite for a remove.

3. Reenumerate the device

   If the device is reenumerated after drivers have deleted their device objects, the PnP manager calls the drivers' *AddDevice* routines and issues an **IRP_MN_START_DEVICE** to reinstate the device. (Also see the Device States from the PnP Perspective figure.)

4. Cancel a query remove

   The PnP manager issues an **IRP_MN_CANCEL_REMOVE_DEVICE** to cancel a query-remove request.

   In response to an **IRP_MN_CANCEL_REMOVE_DEVICE**, the drivers return the device to its started state.

5. Surprise remove (Windows 2000 and later versions of Windows)

   On Windows 2000 and later systems, if a user unplugs a device from the machine without using the Unplug or Eject Hardware program, the PnP manager sends an **IRP_MN_SURPRISE_REMOVAL** IRP.

   This case is called "surprise" removal because the drivers receive no advance warning.

   In response to an **IRP_MN_SURPRISE_REMOVAL** IRP, the drivers for the device fail any outstanding I/O and release the hardware resources used by the device. The drivers must ensure that no components attempt to access the device because it is no longer present.

   All drivers must handle an **IRP_MN_SURPRISE_REMOVAL** IRP and must set status to STATUS_SUCCESS.

   An **IRP_MN_SURPRISE_REMOVAL** cannot be canceled.

6. Remove after surprise remove (Windows 2000 and later versions of Windows)

   When all open handles to the device are closed, the PnP manager sends an **IRP_MN_REMOVE_DEVICE** request to the drivers for the device. Each driver detaches from the device stack and deletes its device object.

7. Surprise remove (Windows 98/Me)

   On Windows 98/Me, a driver does not receive an **IRP_MN_SURPRISE_REMOVAL** when a device is removed without warning. The PnP manager sends only an **IRP_MN_REMOVE_DEVICE**. WDM drivers must have code to handle both an **IRP_MN_SURPRISE_REMOVAL** followed by an **IRP_MN_REMOVE_DEVICE** (the Windows 2000 and later behavior for surprise removal) and an **IRP_MN_REMOVE_DEVICE** without a prior surprise-remove IRP (the Windows 98/Me behavior).

8. Remove after a failed start (Windows 2000 and later)

   If one of the drivers for a device fails an **IRP_MN_START_DEVICE**, the PnP manager sends an **IRP_MN_REMOVE_DEVICE** request to the device stack. Such a remove IRP ensures that all drivers for the device are notified that the device was not successfully started. In response to the **IRP_MN_REMOVE_DEVICE** IRP, the drivers for the device undo their start operations (if they succeeded the start IRP) and undo their *AddDevice* operations. The PnP manager marks such a device as "failed start."

   This behavior applies to Windows 2000 and later platforms only. On Windows 98/Me, the PnP manager sends an **IRP_MN_STOP_DEVICE** in response to a failed start.

A driver for a PnP device can receive an **IRP_MN_SURPRISE_REMOVAL** in more situations than those shown in the figure illustrating typical remove IRP transitions. For example, a user could insert a PC Card into the machine and then remove it before the device is started. In that case, the PnP manager issues a surprise-remove IRP after the drivers' *AddDevice* routines are called but before issuing the **IRP_MN_START_DEVICE** request. A driver for a PnP device must be prepared to handle remove IRPs at any time after the driver's *AddDevice* routine is called.

Handling an IRP_MN_QUERY_REMOVE_DEVICE Request

6/25/2019 • 4 minutes to read • Edit Online

The PnP manager sends this IRP to inform drivers that a device is about to be removed from the machine and to ask whether the device can be removed without disrupting the machine. It also sends this IRP when a user requests to update drivers for the device.

The PnP manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

It does the following before sending this IRP to the drivers for a device:

- Notifies all user-mode applications that registered for notification on the device (or a related device).

  This includes applications that registered for notification on the device, on one of the device's descendants (child device, child of child, and so forth), or on one of the device's removal relations. An application registers for such notification by calling **RegisterDeviceNotification**.

  In response to this notification, an application either prepares for device removal (closes handles to the device) or fails the query.

- Notifies all kernel-mode drivers that registered for notification on the device (or a related device).

  This includes drivers that registered for notification on the device, on one of the device's descendants, or on one of the device's removal relations. A driver registers for this notification by calling **IoRegisterPlugPlayNotification** with an event category of **EventCategoryTargetDeviceChange**.

  In response to this notification, a driver either prepares for device removal (closes handles to the device) or fails the query.

- Sends **IRP_MN_QUERY_REMOVE_DEVICE** IRPs to the drivers for the device's descendants.

- (Windows 2000 and later systems) If a file system is mounted on the device, the PnP manager sends a query-remove request to the file system and any file system filters. If there are open handles to the device, the file system typically fails the query-remove request. If not, the file system typically locks the volume to prevent future creates from succeeding. If a mounted file system does not support a query-remove request, the PnP manager fails the query-remove request for the device.

If all of the above steps succeed, the PnP manager sends the **IRP_MN_QUERY_REMOVE_DEVICE** to the drivers for the device.

An **IRP_MN_QUERY_REMOVE_DEVICE** request is handled first by the top driver in the device stack and then by each next lower driver. A driver handles remove IRPs in its *DispatchPnP* routine.

In response to an **IRP_MN_QUERY_REMOVE_DEVICE**, a driver must do the following:

1. Determine whether the device can be removed from the machine without disrupting operation.

   A driver must fail a query-remove IRP if any of the following are true:

   - If removing the device could result in losing data.

   - If a component has an open handle to the device. (This is an issue on Windows 98/Me only. Windows 2000 and later versions of Windows track open handles and fail the query if there are open handles after the **IRP_MN_QUERY_REMOVE_DEVICE** completes.)

- If a driver has been notified (through an **IRP_MN_DEVICE_USAGE_NOTIFICATION** IRP) that the device is in the path for a paging, crash dump, or hibernation file.
- If the driver has an outstanding interface reference against the device. That is, the driver provided an interface in response to an **IRP_MN_QUERY_INTERFACE** request and the interface has not been dereferenced.

2. If the device cannot be removed, fail the query-remove IRP.

   Set **Irp->IoStatus.Status** to an appropriate error status (typically STATUS_UNSUCCESSFUL), call **IoCompleteRequest** with IO_NO_INCREMENT, and return from the driver's *DispatchPnP* routine. Do not pass the IRP to the next lower driver.

3. If the driver previously sent an **IRP_MN_WAIT_WAKE** request to enable the device for wake-up, cancel the wait-wake IRP.

4. Record the previous PnP state of the device.

   A driver should record the PnP state that the device was in when the driver received the **IRP_MN_QUERY_REMOVE_DEVICE** request because the driver must return the device to that state if the query is canceled (**IRP_MN_CANCEL_REMOVE_DEVICE**). The previous state is typically "started", which is the state that the device enters when the driver successfully completes an **IRP_MN_START_DEVICE** request.

   However, other previous states are possible. For example, the user might have disabled the device through Device Manager. Or, in response to an **IRP_MN_QUERY_CAPABILITIES** request, the parent bus driver (or a filter driver on the bus driver) might have reported that the device's hardware is disabled. In either case, the driver for the disabled device can receive an **IRP_MN_QUERY_REMOVE_DEVICE** request before it receives an **IRP_MN_START_DEVICE** request.

5. Finish the IRP:

   In a function or filter driver:

   - Set **Irp->IoStatus.Status** to STATUS_SUCCESS.
   - Set up the next stack location with **IoSkipCurrentIrpStackLocation** and pass the IRP to the next lower driver with **IoCallDriver**.
   - Propagate the status from **IoCallDriver** as the return status from the *DispatchPnP* routine.
   - Do not complete the IRP.

   In a bus driver:

   - Set **Irp->IoStatus.Status** to STATUS_SUCCESS.
   - Complete the IRP (**IoCompleteRequest**) with IO_NO_INCREMENT.
   - Return from the *DispatchPnP* routine.

If any driver in the device stack fails an **IRP_MN_QUERY_REMOVE_DEVICE**, the PnP manager sends an **IRP_MN_CANCEL_REMOVE_DEVICE** to the device stack. This prevents drivers from requiring an *IoCompletion* routine for a query-remove IRP to detect whether a lower driver failed the IRP.

Once a driver succeeds an **IRP_MN_QUERY_REMOVE_DEVICE** and it considers the device to be in the remove-pending state, the driver must fail any subsequent create requests for the device. The driver processes all other IRPs as usual, until the driver receives an **IRP_MN_CANCEL_REMOVE_DEVICE** or an **IRP_MN_REMOVE_DEVICE**.

Handling an IRP_MN_REMOVE_DEVICE Request

12/7/2018 • 2 minutes to read • Edit Online

The PnP manager uses this IRP to direct drivers to remove a device's software representation (device objects, and so forth). The PnP manager sends this IRP when a device has been removed in an orderly fashion (for example, initiated by a user in the Unplug or Eject Hardware program), by surprise (a user pulls the device from its slot without prior warning), or when the user requests to update drivers.

On Windows 2000 and later systems, the PnP manager sends this IRP when Device Manager disables the device. On Windows 98/Me, the PnP manager sends stop IRPs instead. See Stopping a Device for details.

The PnP manager does the following before sending this IRP to the drivers for a device:

- Sends **IRP_MN_REMOVE_DEVICE** requests to the device's children, if any.
- Notifies any user-mode components and kernel-mode drivers that registered for notification that the device is being removed. The PnP manager calls any user-mode components that registered for target device notification on a handle to the device and calls any kernel-mode drivers that registered for **EventCategoryTargetDeviceChange**.
- (On Windows 2000 and later systems) If a file system is mounted on the device, the PnP manager sends a remove request to the file system and any file system filters. In response, a file system typically dismounts the volume.

The top driver in a device stack handles a remove IRP and passes it to the next lower driver. The parent bus driver for a device is the last driver to perform its remove-device operations. A driver handles remove IRPs in its *DispatchPnP* routine.

Before a driver returns success for an **IRP_MN_REMOVE_DEVICE** request, it must ensure that all resources for the device have been released. This IRP could be the last call before the driver is unloaded.

Removing one device can create the need to remove a series of other devices. The PnP manager coordinates the removal of the additional device objects from the top level down to the root-device level.

This section describes:

Removing a Device in a Function Driver

Removing a Device in a Filter Driver

Removing a Device in a Bus Driver

Removing a Device in a Function Driver

7/9/2019 • 3 minutes to read • Edit Online

When removing a device, a function driver must undo any operations it performed to add and start the device. This discussion includes function drivers for peripheral devices and function drivers for bus devices.

A function driver removes a device using a procedure such as the following in its *DispatchPnP* routine:

1. Is this a function driver for a bus device?

   If so, possibly delete any outstanding child PDOs for devices on the bus.

   If the bus driver handled a previous **IRP_MN_SURPRISE_REMOVAL** request for the child device, but the driver has not yet received the subsequent **IRP_MN_REMOVE_DEVICE** request, the bus driver leaves the child PDO intact. At some later time, when all handles to the child device are closed, the PnP manager will send the remove IRP for the child device and the bus driver deletes the child PDO at that time.

   If the bus driver handled a previous **IRP_MN_REMOVE_DEVICE** request for the device, and there has been no subsequent **IRP_MN_SURPRISE_REMOVAL** request, then the bus driver deletes the child PDO. In this case, the PnP manager ensures that any function and filter drivers have been removed from the child device (FDO and filter DOs have been deleted) before it sends a remove IRP to the parent bus device. The child PDO might still be present, so the bus driver must delete the child PDO before it removes the bus device.

2. Has the driver already handled a previous **IRP_MN_SURPRISE_REMOVAL** request for this FDO?

   If so, perform any remaining clean-up and skip to step 8, **IoCallDriver**.

   A driver typically maintains a flag in the device extension that indicates whether the driver has handled an **IRP_MN_SURPRISE_REMOVAL** request for the device.

3. If the driver previously enabled the device for wake-up, cancel the **IRP_MN_WAIT_WAKE** request.

4. Ensure that the device is inactive.

   If the device is not already inactive in response to a prior **IRP_MN_QUERY_REMOVE_DEVICE**, the driver must mark the device as not accepting new requests and must complete any requests queued in this driver. The driver must fail any outstanding requests that require access to the device.

   A driver can use the **Io*Xxx*RemoveLock*Xxx*** routines to count outstanding I/O and to set an event indicating that remove processing can continue.

5. Perform any power-down operations.

   Each driver for the device performs its power-down operations, if any, when it receives the **IRP_MN_REMOVE_DEVICE** request. The power policy owner for the device, typically the function driver, does not send a separate **IRP_MN_SET_POWER** request to set the device power state to D3. The parent bus driver typically powers down the slot and notifies the power manager with **PoSetPowerState** when the bus driver gets the remove IRP. For additional information, see Power Management.

6. Disable any device interfaces by calling **IoSetDeviceInterfaceState**.

7. Free any hardware resources for the device in use by the driver.

   The exact operations depend on the device and the driver but can include disconnecting an interrupt with **IoDisconnectInterrupt**, freeing physical address ranges with **MmUnmapIoSpace**, and freeing I/O ports.

8. Pass the **IRP_MN_REMOVE_DEVICE** request down to the next driver.

   Set up the IRP stack location for the next lower driver with **IoSkipCurrentIrpStackLocation** and pass the IRP to the next driver with **IoCallDriver**.

   A driver is not required to wait for underlying drivers to finish their remove operations before continuing with its remove activities.

9. Remove the device object from the device stack with **IoDetachDevice**.

   Specify a pointer to the next lower device object as the *TargetDevice* parameter. The driver receives such a pointer from the call to **IoAttachDeviceToDeviceStack** in the driver's *AddDevice* routine.

10. Clean up any device-specific allocations, memory, events, and so forth.

11. Free the FDO with **IoDeleteDevice**.

12. Return from the *DispatchPnP* routine, propagating the return status from **IoCallDriver**.

A function driver does not specify an *IoCompletion* routine for a remove IRP, nor does it complete the IRP. Remove IRPs are completed by the parent bus driver.

Removing a Device in a Filter Driver

12/5/2018 • 2 minutes to read • Edit Online

When removing a device, a filter driver must undo any operations it performed to add and start the device. A filter driver follows essentially the same procedure as a function driver when removing a device.

Removing a Device in a Bus Driver

6/25/2019 • 3 minutes to read • Edit Online

When removing a child device (child PDO), the parent bus driver must undo any operations it performed to add and start the device.

A bus driver removes a child device with a procedure such as the following in its *DispatchPnP* routine:

1. Has the driver handled a previous **IRP_MN_SURPRISE_REMOVAL** request for this PDO?

   If so, perform any remaining clean-up and skip to step 4.

   A driver typically maintains a flag in the device extension that indicates whether the driver has handled an **IRP_MN_SURPRISE_REMOVAL** request for the device.

2. Complete any requests queued in the driver.

3. Remove power from the device, if the bus driver is capable of doing so, and notify the power manager by calling **PoSetPowerState**.

   The bus driver powers down the child device, if possible, and notifies the power manager of the device's change in power state. The bus driver does this in response to the **IRP_MN_REMOVE_DEVICE** request; the device's power policy owner does not send an **IRP_MN_SET_POWER** request when the device is being removed. For additional information, see Power Management.

4. If the bus driver reported this device in its most recent response to an **IRP_MN_QUERY_DEVICE_RELATIONS** request for **BusRelations**, the device is still physically present on the machine. In this case, the bus driver:

   - Retains the PDO for the device until the device has been physically removed.
   - Sets **Irp->IoStatus.Status** to STATUS_SUCCESS.
   - Completes the IRP with **IoCompleteRequest**.
   - Returns from the *DispatchPnP* routine.

   The bus driver must continue to report this device in subsequent enumerations (**IRP_MN_QUERY_DEVICE_RELATIONS** for **BusRelations**) until the device is physically removed. The PnP manager keeps track of whether an enumerated device has been added and started.

5. If the device was not included in the bus driver's most recent response to an **IRP_MN_QUERY_DEVICE_RELATIONS** request for **BusRelations**, the bus driver considers the device to be physically removed from the machine. In this case, the bus driver does the following:

   - Cleans up device-specific allocations, memory, events, and so forth.
   - Sets **Irp->IoStatus.Status** to STATUS_SUCCESS.
   - Completes the IRP with **IoCompleteRequest**.
   - Frees the PDO with **IoDeleteDevice**.

     The bus driver must delete the PDO if the driver omitted the device from its most recent **BusRelations** list. If a user plugs the device into the machine again, the bus driver must create a new PDO in response to the next **BusRelations** query. If a bus driver reuses the same PDO for a new instance of a device, the machine will not operate properly.

- Returns from the *DispatchPnP* routine.

If the device is still present when the PnP manager sends the **IRP_MN_REMOVE_DEVICE** request, the bus driver retains the PDO. If, at some later time, the device is physically removed from the bus, the PnP manager sends another **IRP_MN_REMOVE_DEVICE**. Upon receipt of the subsequent remove IRP, the bus driver deletes the PDO for the device.

A bus driver must be able to handle an **IRP_MN_REMOVE_DEVICE** for a device it has already removed and whose PDO is marked for deletion. In response to such an IRP, the bus driver can succeed the IRP or return STATUS_NO_SUCH_DEVICE. The PDO for the device has not yet been deleted in this case, despite the bus driver's previous call to **IoDeleteDevice**, because some component still has a reference to the object. Therefore, the bus driver can access the PDO while handling the second remove IRP. The bus driver must not call **IoDeleteDevice** a second time for the PDO; the I/O system deletes the PDO when its reference count reaches zero.

A bus driver does not remove its data structures for a child device until it receives an **IRP_MN_REMOVE_DEVICE** request for the device. A bus driver might detect that a device has been removed and call **IoInvalidateDeviceRelations**, but it must not delete the device's PDO until the PnP manager sends an **IRP_MN_REMOVE_DEVICE** request.

Handling an IRP_MN_CANCEL_REMOVE_DEVICE Request

6/25/2019 • 2 minutes to read • Edit Online

In response to an **IRP_MN_CANCEL_REMOVE_DEVICE** request, the drivers for a device must return the device to the state it was in prior to receiving the **IRP_MN_QUERY_REMOVE_DEVICE** request. Typically, drivers return the device to the started state.

In addition to sending an **IRP_MN_CANCEL_REMOVE_DEVICE** to a device, the PnP manager sends the IRP to the device's removal relations, if any. The PnP manager also sends a cancel-remove IRP to the device's children.

The PnP manager calls any **EventCategoryTargetDeviceChange** notification callbacks after the **IRP_MN_CANCEL_REMOVE_DEVICE** request completes. Such callbacks were registered on the device by calling **IoRegisterPlugPlayNotification**. The PnP manager also calls any user-mode components that registered for such notification by calling **RegisterDeviceNotification**.

An **IRP_MN_CANCEL_REMOVE_DEVICE** request must be handled first by the parent bus driver for a device and then by each higher driver in the device stack. A driver handles remove IRPs in its *DispatchPnP* routine.

A driver handles an **IRP_MN_CANCEL_REMOVE_DEVICE** request with a procedure such as the following in its *DispatchPnP* routine:

1. In a function or filter driver, postpone restarting the device until lower drivers have completed their restart operations.

   A function or filter driver sets an *IoCompletion* routine, passes the **IRP_MN_CANCEL_REMOVE_DEVICE** down the device stack, and postpones its restart operations until all lower drivers have finished with the IRP. (See Postponing PnP IRP Processing Until Lower Drivers Finish.)

2. After lower drivers finish, return the device to its previous PnP state.

   The drivers return the device to the state it was in prior to receiving the **IRP_MN_QUERY_REMOVE_DEVICE** request. Typically, drivers return the device to the started state. Exact operations depend on the device and the driver.

   If the device was previously enabled for wake-up, the device power policy owner (typically the function driver) should send an **IRP_MN_WAIT_WAKE** request to reenable wake-up. See Power Management for details.

3. Set **Irp->IoStatus.Status** to STATUS_SUCCESS and complete the IRP with **IoCompleteRequest**.

   As with any PnP IRP, a bus driver completes the IRP.

   A function or filter driver also completes the IRP, in this case because the driver's *IoCompletion* routine interrupted completion processing by returning STATUS_MORE_PROCESSING_REQUIRED.

   Drivers must succeed this IRP. If any driver fails this IRP, the device is left in an inconsistent state.

A driver might receive a spurious cancel-remove request when the device is started and active. This can occur, for example, if the driver (or a driver higher in the device stack) failed an **IRP_MN_QUERY_REMOVE_DEVICE** request. When a device is started and active, a driver simply succeeds a spurious cancel-remove request for the device.

Handling an IRP_MN_SURPRISE_REMOVAL Request

6/25/2019 • 8 minutes to read • Edit Online

The Windows 2000 and later PnP manager sends this IRP to notify drivers that a device is no longer available for I/O operations and has probably been unexpectedly removed from the machine ("surprise removal").

The PnP manager sends an **IRP_MN_SURPRISE_REMOVAL** request for the following reasons:

- If the bus has hot-plug notification, it notifies the device's parent bus driver that the device has disappeared. The bus driver calls **IoInvalidateDeviceRelations**. In response, the PnP manager queries the bus driver for its children (**IRP_MN_QUERY_DEVICE_RELATIONS** for **BusRelations**). The PnP manager determines that the device is not in the new list of children and initiates its surprise-removal operations for the device.
- The bus is enumerated for another reason and the surprise-removed device is not included in the list of children. The PnP manager initiates its surprise removal operations.
- The function driver for the device determines that the device is no longer present (because, for example, its requests repeatedly time out). The bus might be enumerable but it does not have hot-plug notification. In this case, the function driver calls **IoInvalidateDeviceState**. In response, the PnP manager sends an **IRP_MN_QUERY_PNP_DEVICE_STATE** request to the device stack. The function driver sets the PNP_DEVICE_FAILED flag in the **PNP_DEVICE_STATE** bitmask indicating that the device has failed.
- The driver stack successfully completes an **IRP_MN_STOP_DEVICE** request but then fails a subsequent **IRP_MN_START_DEVICE** request. In such cases, the device is probably still connected.

All PnP drivers must handle this IRP and must set **Irp->IoStatus.Status** to STATUS_SUCCESS. A driver for a PnP device must be prepared to handle **IRP_MN_SURPRISE_REMOVAL** at any time after the driver's *AddDevice* routine is called. Proper handling of the IRP enables the drivers and the PnP manager to:

1. Disable the device, in case it is still connected.

   If the driver stack successfully completed an **IRP_MN_STOP_DEVICE** request but then, for some reason, failed a subsequent **IRP_MN_START_DEVICE** request, the device must be disabled.

2. Release hardware resources assigned to the device and make them available to another device.

   As soon as a device is no longer available, its hardware resources should be freed. The PnP manager can then reassign the resources to another device, including the same device, which a user might hot-plug back into the machine.

3. Minimize the risk of data loss and system disruption.

   Devices that support hot-plugging and their drivers should be designed to handle surprise removal. Users expect to be able to remove devices that support hot-plugging at any time.

The PnP manager sends an **IRP_MN_SURPRISE_REMOVAL** at IRQL = PASSIVE_LEVEL in the context of a system thread.

The PnP manager sends this IRP to drivers before notifying user-mode applications and other kernel-mode components. After the IRP completes, the PnP manager sends an **EventCategoryTargetDeviceChange** notification with GUID_TARGET_DEVICE_REMOVE_COMPLETE to kernel-mode components that registered for such notification on the device.

The **IRP_MN_SURPRISE_REMOVAL** IRP is handled first by the top driver in the device stack and then by each

next lower driver.

In response to **IRP_MN_SURPRISE_REMOVAL**, a driver must do the following, in the listed order:

1. Determine if the device has been removed.

   The driver must always attempt to determine if the device is still connected. If it is, the driver must attempt to stop the device and disable it.

2. Release the device's hardware resources (interrupts, I/O ports, memory registers, and DMA channels).

3. In a parent bus driver, power down the bus slot if the driver is capable of doing so. Call **PoSetPowerState** to notify the power manager. For additional information, see Power Management.

4. Prevent any new I/O operations on the device.

   A driver should process subsequent **IRP_MJ_CLEANUP**, **IRP_MJ_CLOSE**, **IRP_MJ_POWER**, and **IRP_MJ_PNP** requests, but the driver must prevent any new I/O operations. A driver must fail any subsequent IRPs that the driver would have handled if the device were present, besides close, clean-up, and PnP IRPs.

   A driver can set a bit in the device extension to indicate that the device has been surprise-removed. The driver's dispatch routines should check this bit.

5. Fail outstanding I/O requests on the device.

6. Continue to pass down any IRPs that the driver does not handle for the device.

7. Disable device interfaces with **IoSetDeviceInterfaceState**.

8. Clean up any device-specific allocations, memory, events, or other system resources.

   A driver could postpone such clean-up until it receives the subsequent **IRP_MN_REMOVE_DEVICE** request, but if a legacy component has an open handle that cannot be closed, the remove IRP will never be sent.

9. Leave the device object attached to the device stack.

   Do not detach and delete the device object until the subsequent **IRP_MN_REMOVE_DEVICE** request.

10. Finish the IRP.

    In a function or filter driver:

    - Set **Irp->IoStatus.Status** to STATUS_SUCCESS.
    - Set up the next stack location with **IoSkipCurrentIrpStackLocation** and pass the IRP to the next lower driver with **IoCallDriver**.
    - Propagate the status from **IoCallDriver** as the return status from the *DispatchPnP* routine.
    - Do not complete the IRP.

    In a bus driver (that is handling this IRP for a child PDO):

    - Set **Irp->IoStatus.Status** to STATUS_SUCCESS.
    - Complete the IRP (**IoCompleteRequest**) with IO_NO_INCREMENT.
    - Return from the *DispatchPnP* routine.

After this IRP succeeds and all open handles to the device are closed, the PnP manager sends an **IRP_MN_REMOVE_DEVICE** request to the device stack. In response to the remove IRP, drivers detach their

device objects from the stack and delete them. If a legacy component has a handle open to the device and it leaves the handle open despite I/O failures, the PnP manager never sends the remove IRP.

All drivers should handle this IRP and should note that the device has been physically removed from the machine. Some drivers, however, will not cause adverse results if they do not handle the IRP. For example, a device that consumes no system hardware resources and resides on a protocol-based bus, such as USB or 1394, cannot tie up hardware resources because it does not consume any. There is no risk of drivers attempting to access the device after it has been removed because the function and filter drivers access the device only through the parent bus driver. Because the bus supports removal notification, the parent bus driver is notified when the device disappears and the bus driver fails all subsequent attempts to access the device.

On Windows 98/Me, the PnP manager does not send this IRP. If a user removes a device without first using the appropriate user interface, the PnP manager sends only an **IRP_MN_REMOVE_DEVICE** request to the drivers for the device. All WDM drivers must handle both **IRP_MN_SURPRISE_REMOVAL** and **IRP_MN_REMOVE_DEVICE**. The code for **IRP_MN_REMOVE_DEVICE** should check whether the driver received a prior surprise-remove IRP and should handle both cases.

Using GUID_REENUMERATE_SELF_INTERFACE_STANDARD

The GUID_REENUMERATE_SELF_INTERFACE_STANDARD interface enables a driver to request that its device be reenumerated.

To use this interface, send an IRP_MN_QUERY_INTERFACE IRP to your bus driver with InterfaceType = GUID_REENUMERATE_SELF_INTERFACE_STANDARD. The bus driver supplies a pointer to a REENUMERATE_SELF_INTERFACE_STANDARD structure that contains pointers to the individual routines of the interface. A ReenumerateSelf routine requests that a bus driver reenumerate a child device.

About PNP_DEVICE_STATE

The PNP_DEVICE_STATE type is a bitmask that describes the PnP state of a device. A driver returns a value of this type in response to an **IRP_MN_QUERY_PNP_DEVICE_STATE** request.

```
typedef ULONG PNP_DEVICE_STATE, *PPNP_DEVICE_STATE;
```

The flag bits in a PNP_DEVICE_STATE value are defined as follows.

FLAG BIT	DESCRIPTION
PNP_DEVICE_DISABLED	The device is physically present but is disabled in hardware.
PNP_DEVICE_DONT_DISPLAY_IN_UI	Do not display the device in the user interface. Set for a device that is physically present but not usable in the current configuration, such as a game port on a laptop that is not usable when the laptop is undocked. (Also see the **NoDisplayInUI** flag in the **DEVICE_CAPABILITIES** structure.)

FLAG BIT	DESCRIPTION
PNP_DEVICE_FAILED	The device is present but not functioning properly. When both this flag and PNP_DEVICE_RESOURCE_REQUIREMENTS_CHANGED are set, the device must be stopped before the PnP manager assigns new hardware resources (nonstop rebalance is not supported for the device).
PNP_DEVICE_NOT_DISABLEABLE	The device is required when the computer starts. Such a device must not be disabled. A driver sets this bit for a device that is required for proper system operation. For example, if a driver receives notification that a device is in the paging path (**IRP_MN_DEVICE_USAGE_NOTIFICATION** for **DeviceUsageTypePaging**), the driver calls **IoInvalidateDeviceState** and sets this flag in the resulting **IRP_MN_QUERY_PNP_DEVICE_STATE** request. If this bit is set for a device, the PnP manager propagates this setting to the device's parent device, its parent's parent device, and so forth. If this bit is set for a root-enumerated device, the device cannot be disabled or uninstalled.
PNP_DEVICE_REMOVED	The device has been physically removed.
PNP_DEVICE_RESOURCE_REQUIREMENTS_CHANGED	The resource requirements for the device have changed. Typically, a bus driver sets this flag when it has determined that it must expand its resource requirements in order to enumerate a new child device.
PNP_DEVICE_DISCONNECTED	The device driver is loaded, but this driver has detected that the device is no longer connected to the computer. Typically, this flag is used for function drivers that communicate with wireless devices. For example, the flag is set when the device moves out of range, and is cleared after the device moves back into range and re-connects. A bus driver does not typically set this flag. The bus driver should instead stop enumerating the child device if the device is no longer connected. This flag is used only if the function driver manages the connection. The sole purpose of this flag is to let clients know whether the device is connected. Setting the flag does not affect whether the driver is loaded.

The PnP manager queries a device's PNP_DEVICE_STATE right after starting the device by sending an **IRP_MN_QUERY_PNP_DEVICE_STATE** request to the device stack. In response to this IRP, the drivers for the device set the appropriate flags in PNP_DEVICE_STATE.

If any of the state characteristics change after the initial query, a driver notifies the PnP manager by calling **IoInvalidateDeviceState**. In response to a call to **IoInvalidateDeviceState**, the PnP manager queries the device's PNP_DEVICE_STATE again.

If a device is marked PNP_DEVICE_NOT_DISABLEABLE, the debugger displays a DNUF_NOT_DISABLEABLE user flag for the devnode. The debugger also displays a **DisableableDepends** value that counts the number of reasons why the device cannot be disabled. This value is the sum of X+Y, where X is one if the device cannot be disabled and Y is the count of the device's child devices that cannot be disabled.

Using Remove Locks

6/25/2019 • 2 minutes to read • Edit Online

The remove lock routines provide a way to track the number of outstanding I/O operations on a device, and to determine when it is safe to detach and delete a driver's device object. The system provides these routines to driver writers as an alternative to implementing their own tracking mechanism.

A driver can use this mechanism for two purposes:

1. To ensure that the driver's *DispatchPnP* routine will not complete an **IRP_MN_REMOVE_DEVICE** request while the lock is held (for example, while another driver routine is accessing the device).
2. To count the number of reasons why the driver should not delete its device object, and to set an event when that count goes to zero.

To initialize a remove lock, a driver should allocate an **IO_REMOVE_LOCK** structure in its device extension and then call **IoInitializeRemoveLock**. A driver typically calls **IoInitializeRemoveLock** in its *AddDevice* routine, when the driver initializes the rest of the device extension for a device object.

Your driver must call **IoAcquireRemoveLock** each time it starts an I/O operation. The driver must call **IoReleaseRemoveLock** each time it finishes an I/O operation. A driver can acquire the lock more than once. The remove lock routines maintain a count of the outstanding acquisitions of the lock. Each call to **IoAcquireRemoveLock** increments the count, and **IoReleaseRemoveLock** decrements the count.

Your driver should also call **IoAcquireRemoveLock** when it passes out a reference to its code (for timers, DPCs, callbacks, and so on). The driver then must call **IoReleaseRemoveLock** when the event has returned.

In its dispatch code for **IRP_MN_REMOVE_DEVICE**, the driver must acquire the lock once more and then call **IoReleaseRemoveLockAndWait**. This routine does not return until all outstanding acquisitions of the lock have been released. To allow queued I/O operations to complete, each driver should call **IoReleaseRemoveLockAndWait** *after* it passes the **IRP_MN_REMOVE_DEVICE** request to the next-lower driver, and *before* it releases memory, calls **IoDetachDevice**, or calls **IoDeleteDevice**. After **IoReleaseRemoveLockAndWait** has been called for a particular remove lock, all subsequent calls to **IoAcquireRemoveLock** for the same remove lock will fail.

After **IoReleaseRemoveLockAndWait** returns, the driver should consider the device to be in a state in which it is ready to be removed and cannot perform I/O operations. Therefore, the driver must not call **IoInitializeRemoveLock** to re-initialize the remove lock. Violation of this rule while the driver is being verified by Driver Verifier will result in a bug check.

Because a driver stores an **IO_REMOVE_LOCK** structure in the device extension of a device object, the remove lock is deleted when the driver deletes the device extension while processing an **IRP_MN_REMOVE_DEVICE** request.

Using PnP Notification

12/5/2018 • 2 minutes to read • Edit Online

In a PnP environment, drivers and applications need to react to changes in the configuration of devices on the machine. For example, an application needs to know when a device of interest has been added to the machine and a driver needs to know when a change occurs on a particular device.

The PnP manager provides a mechanism for drivers and applications to be notified when certain PnP events occur. This section describes how to use PnP notification in kernel-mode code. Writers of user-mode applications should see the Microsoft Windows SDK documentation For information about the **RegisterDeviceNotification** function and related functions.

PnP Notification Overview

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager provides a mechanism for drivers and applications to be notified when certain events occur on a specific device or on the system in general. A driver can register for notification of the following categories of events:

- **EventCategoryDeviceInterfaceChange**

  When a driver registers for this category of events on a device interface, the PnP manager notifies the driver of the following events:

  GUID_DEVICE_INTERFACE_ARRIVAL
  Indicates that a device interface of the specified class has been enabled. For example, a user added a new disk to the machine and the volume manager enabled a new volume (a device interface of the class "volume").

  GUID_DEVICE_INTERFACE_REMOVAL
  Indicates that a device interface of the specified class has been disabled.

  See **IoRegisterDeviceInterface** and related routines for more information about device interfaces.

- **EventCategoryTargetDeviceChange**

  When a driver registers for this category of events on a device, the PnP manager notifies the driver when the following events occur on the device:

  GUID_TARGET_DEVICE_QUERY_REMOVE
  Indicates that the PnP manager is about to remove the drivers for the device. Several actions can cause this event, including: a user has requested to remove the specified device from the machine or a user has issued an update-driver request for the device. This notification requests the drivers for the device to either approve or veto the impending remove operation.

  GUID_TARGET_DEVICE_REMOVE_COMPLETE
  Indicates that the specified device has been removed from the machine or that a user is changing the driver(s) for the device.

  GUID_TARGET_DEVICE_REMOVE_CANCELLED
  Indicates that an impending remove operation on the specified device has been canceled.

  GUID_*XXX* (custom events)
  Indicates that a custom event has occurred on the specified device.

  A driver writer can define a custom event for a device. When the driver (or another related component) notifies the PnP manager that the custom event has occurred, the PnP manager notifies any components that registered for target device change notifications on the device.

  Unlike registering for device interface changes, which can be considered a "passive" interest in the interface, registering for target device changes indicates an "active" interest in a device.

- **EventCategoryHardwareProfileChange**

  This category includes the following events:

  GUID_HWPROFILE_QUERY_CHANGE

Indicates that a user has requested to change the hardware profile of the machine. The PnP manager uses this notification to ask registered components whether it can change the hardware profile without disrupting system operation. Registered components typically succeed these query requests.

GUID_HWPROFILE_CHANGE_COMPLETE
Indicates that the hardware profile of the machine has changed. If a driver maintains profile-specific settings, such a driver should refresh those settings after a hardware profile change.

GUID_HWPROFILE_CHANGE_CANCELLED
Indicates that an impending hardware profile change has been canceled.

PnP notification works as follows for kernel-mode components:

1. A driver registers for notification on a category of events by calling **IoRegisterPlugPlayNotification**.

   A PnP notification callback routine remains registered until the driver explicitly removes the registration.

2. The PnP manager calls the driver's callback routine when an event in the registered category occurs.

3. The driver removes the callback registration by calling **IoUnregisterPlugPlayNotification**.

Drivers must not generate a synchronous event or wait for an asynchronous event to occur during the processing of a close.

For further information about PnP notification, see the following sections:

Guidelines for Writing PnP Notification Callback Routines

Using PnP Device Interface Change Notification

Using PnP Target Device Change Notification

Using PnP Hardware Profile Change Notification

Using PnP Custom Notification

Guidelines for Writing PnP Notification Callback Routines

6/25/2019 • 2 minutes to read • Edit Online

The PnP manager calls notification callback routines at IRQL = PASSIVE_LEVEL.

To ensure smooth operation of the PnP subsystem, a PnP notification callback routine must follow these guidelines:

1. A notification callback routine must not block.
2. A notification callback routine must not call, or cause a call to, synchronous routines that generate PnP events or any routine that blocks waiting for device installation or removal.

   Calling such routines during a notification callback can cause a system deadlock.

   For example, a driver must not call **IoReportTargetDeviceChange** in a notification callback routine. Call **IoReportTargetDeviceChangeAsynchronous** instead.
3. A notification callback routine should return success for any events it does not explicitly fail.

   When a driver registers for notification on an event category, the PnP manager notifies the driver of all events in that category, present and future. If a driver returns an error status for events it does not handle, the driver risks failing a new query event by mistake.

   A driver correctly returns an error status when, for example, the driver fails a query notification to veto the event being proposed.
4. A notification callback routine should be paged code.

Using PnP Device Interface Change Notification

6/25/2019 • 2 minutes to read • Edit Online

A driver registers for **EventCategoryDeviceInterfaceChange** notification so the driver can be notified when device interfaces of a particular class arrive (are enabled) or are removed (disabled) on the machine. For example, a composite battery driver might register for notification of device interfaces of class battery so it can provide information to the operating system about total available battery power.

The following subsections discuss how to register for device interface change notification and how to handle device interface change events in a PnP notification callback routine:

Registering for Device Interface Change Notification

Handling Device Interface Change Events

See **IoRegisterDeviceInterface** and related routines For information about device interfaces.

Registering for Device Interface Change Notification

6/25/2019 • 2 minutes to read • Edit Online

A driver registers for notification of device interface arrival and removal events by calling **IoRegisterPlugPlayNotification**.

The following information applies to calling this routine for device interface change notification:

- Specify an *EventCategory* of **EventCategoryDeviceInterfaceChange**.
- *EventCategoryData* must point to the GUID for a device interface class.

  The GUID for a interface class is typically defined in a header file with the structures, constants, and so forth, for the interface.
- Specify an *EventCategoryFlags* of PNPNOTIFY_DEVICE_INTERFACE_INCLUDE_EXISTING_INTERFACES.

  This flag directs the PnP manager to register the *CallbackRoutine* for future device interface arrivals and departures of the specified class and to call the *CallbackRoutine* immediately for any relevant device interfaces that are already active.

  A driver can call **IoGetDeviceInterfaces** to get a list of existing interfaces of a specific class and then register its callback routine without this flag, but using the flag is easier and avoids a potential timing issue.
- Specify a driver-defined *Context*, if appropriate, that the PnP manager will pass to the callback routine.

A driver that opens a handle to a device in response to a device interface arrival notification should register for **EventCategoryTargetDeviceChange** events on the device. (See Using PnP Target Device Change Notification.)

A driver cancels notification registration by calling **IoUnregisterPlugPlayNotification** with the *NotificationEntry* returned by **IoRegisterPlugPlayNotification**.

Handling Device Interface Change Events

12/5/2018 • 2 minutes to read • Edit Online

When a driver or a user-mode component enables or disables a device interface instance, the PnP manager calls all notification callback routines that are registered for **EventCategoryDeviceInterfaceChange** events on the device interface class. To indicate the reason for the notification, the PnP manager sets the **Event** member of the callback routine's *NotificationStructure* parameter to GUID_DEVICE_INTERFACE_ARRIVAL or GUID_DEVICE_INTERFACE_REMOVAL.

When handling a GUID_DEVICE_INTERFACE_ARRIVAL event, a notification callback routine should:

- Perform driver-defined tasks for handling the new interface.

  Typically, a notification callback routine directly opens the device in the context of the callback. However, if opening the device can cause subsequent PnP events to occur (for example, the enumeration of child devices), the callback routine should instead queue a worker routine to open the device; otherwise, a deadlock can occur.

  A callback routine might enable an interface of its own in response to the availability of the new interface.

When handling a GUID_DEVICE_INTERFACE_REMOVAL event, a notification callback routine should:

- Undo whatever operations it performed when the interface was enabled.

When the device is removed, the driver should close the file handle that it opened during the GUID_DEVICE_INTERFACE_ARRIVAL event callback. For an orderly device removal, the driver should close the file handle during the GUID_TARGET_DEVICE_QUERY_REMOVE event callback. For a surprise removal, the driver should close the file handle during the GUID_TARGET_DEVICE_REMOVE_COMPLETE event callback. Do not close the file handle during the GUID_DEVICE_INTERFACE_REMOVAL event callback.

Using PnP Target Device Change Notification

12/5/2018 • 2 minutes to read • Edit Online

A driver registers for **EventCategoryTargetDeviceChange** notification on a device so the driver can be notified when the device is about to be removed. For example, if a driver opens a handle to a device, the driver should register for **EventCategoryTargetDeviceChange** notification on the device so the driver can close its handle when the PnP manager needs to remove the device.

Drivers can also use **EventCategoryTargetDeviceChange** notification for custom notification. (See Using PnP Custom Notification.)

> **IMPORTANT**
>
> Registering for PnP target device change notifications is not intended to notify listeners about target device power state changes. If a driver needs to know about a target device power change, the driver should instead define a power relation between devices.
>
> To define a power relation, the driver calls **IoInvalidateDeviceRelations** with the *Type* parameter set to **PowerRelations**, then responds to the PnP manager's IRP_MN_QUERY_DEVICE_RELATIONS query for **PowerRelations** with the correct information.

The following subsections discuss how to register for target device change notification and how to handle target device change events in a PnP notification callback routine:

Registering for Target Device Change Notification

Handling a GUID_TARGET_DEVICE_QUERY_REMOVE Event

Handling a GUID_TARGET_DEVICE_REMOVE_COMPLETE Event

Handling a GUID_TARGET_DEVICE_REMOVE_CANCELLED Event

Registering for Target Device Change Notification

6/25/2019 • 2 minutes to read • Edit Online

A driver registers for notification of PnP target device change events by calling **IoRegisterPlugPlayNotification**.

The following information applies to calling this routine for target device change notification:

- Specify an *EventCategory* of **EventCategoryTargetDeviceChange**.
- *EventCategoryData* must point to the file object for the device on which notification is requested.

  If the driver's callback routine requires access to the file object, the driver should take out a reference on the file object before calling **IoRegisterPlugPlayNotification**.

  If the driver's callback routine does not require access to the file object, the driver does not need to reference the object.

  After the file object is closed, the driver continues to receive notifications for the device until the driver removes its notification registration. This design allows the driver to receive notification of GUID_TARGET_DEVICE_REMOVE_CANCELLED events, for example.
- Specify a driver-defined *Context* that the PnP manager will pass to the callback routine.

  A driver might use the *Context* parameter to maintain information about the current state of the file object (for example, has it been closed/deleted).

  A driver might also use the *Context* to store the path it used to originally open the device. A driver can use this path to reopen the device after a canceled remove operation. (See Handling a GUID_TARGET_DEVICE_REMOVE_CANCELLED Event for more information.)

A driver removes a notification registration by calling **IoUnregisterPlugPlayNotification** with the *NotificationEntry* returned by **IoRegisterPlugPlayNotification**. If the driver took out a reference on the file object when it registered for notification and that reference is still outstanding, the driver must release the reference after it removes the registration.

Handling a GUID_TARGET_DEVICE_QUERY_REMOVE Event

6/25/2019 • 2 minutes to read • Edit Online

Before the PnP manager sends an **IRP_MN_QUERY_REMOVE_DEVICE** IRP to the drivers for a device, it calls any notification callback routines that registered for **EventCategoryTargetDeviceChange** on the device. The PnP manager specifies a *NotificationStructure*.**Event** of GUID_TARGET_DEVICE_QUERY_REMOVE.

In response to such a notification, the callback routine determines whether the device can be removed without disrupting the system.

If the device should not be removed, the callback routine returns STATUS_UNSUCCESSFUL. In response to this status, the PnP manager aborts query-remove processing and the device will not be removed.

If the device can be removed, the callback routine should perform any appropriate operations to prepare for device removal, such as closing any handles open on the device (if possible). If handles remain open on the device, the PnP manager cannot remove the device, and the PnP manager aborts query-remove processing.

When successfully handling a GUID_TARGET_DEVICE_QUERY_REMOVE event, a notification callback routine should:

- Close any open handles to the device.
- If the driver has an outstanding reference on the file object, dereference the file object.
- Remain registered for future **EventCategoryTargetDeviceChange** notifications. This is important because the impending remove operation might be canceled.

Closing a handle to a device does not cancel a driver's registration for PnP target device change notification. The PnP manager can still call the driver's notification callback routine, but in such calls the file object in the *NotificationStructure* is not valid.

Handling a GUID_TARGET_DEVICE_REMOVE_COMPLETE Event

6/25/2019 • 2 minutes to read • Edit Online

Before the PnP manager sends an **IRP_MN_REMOVE_DEVICE** IRP to the drivers for a device, the PnP manager calls any kernel-mode notification callback routines that registered for **EventCategoryTargetDeviceChange** on the device. The PnP manager specifies a *NotificationStructure*.**Event** of GUID_TARGET_DEVICE_REMOVE_COMPLETE.

When handling a GUID_TARGET_DEVICE_REMOVE_COMPLETE event, a notification callback routine should:

- Remove notification registration on the device.

  The device has been removed, so the driver calls **IoUnregisterPlugPlayNotification** to remove the notification registration.

  The device may still be physically present on the machine, but all device objects have been deleted and the device is not available for use.

- Perform surprise-remove processing if the driver did not receive a previous query-remove notification.

  If a device is surprise-removed, the PnP manager sends registered drivers a remove-complete notification without a prior query-remove notification. In this case a driver has to perform any necessary cleanup, such as closing any handles to the device and removing any outstanding references to the file object.

Handling a GUID_TARGET_DEVICE_REMOVE_CANCELLED Event

6/25/2019 • 2 minutes to read • Edit Online

If an **IRP_MN_QUERY_REMOVE_DEVICE** request fails, the PnP manager sends an **IRP_MN_CANCEL_REMOVE_DEVICE** IRP to the drivers for the device. After the cancel-remove IRP completes successfully, the PnP manager calls any notification callback routines that registered for **EventCategoryTargetDeviceChange** on the device. The PnP manager specifies a *NotificationStructure*.**Event** of GUID_TARGET_DEVICE_REMOVE_CANCELLED.

When handling a GUID_TARGET_DEVICE_REMOVE_CANCELLED event, a notification callback routine should:

- Reregister for target device notification.

  Because the driver closed the previous registration handle in response to the query-remove notification, the driver must open a new handle. The driver must:

  1. Remove the old registration with **IoUnregisterPlugPlayNotification**.
  2. Open a new handle to the device.
  3. Reregister for notification on the new handle with **IoRegisterPlugPlayNotification**.

Using PnP Hardware Profile Change Notification

12/5/2018 • 2 minutes to read • Edit Online

A driver registers for **EventCategoryHardwareProfileChange** notification so the driver can be notified when the machine transitions from one hardware profile to another. For example, a driver can use this mechanism to be notified when a laptop is docked or undocked.

The following subsections discuss how to register for hardware profile change notification and how to handle hardware profile change events in a PnP notification callback routine:

Registering for Hardware Profile Change Notification

6/25/2019 • 2 minutes to read • Edit Online

A driver registers for notification of hardware profile changes by calling **IoRegisterPlugPlayNotification**.

The following information applies to calling this routine for hardware profile change notification:

- Specify an *EventCategory* of **EventCategoryHardwareProfileChange**.
- *EventCategoryData* must be **NULL**.
- Specify a driver-defined *Context*, if appropriate, that the PnP manager will pass to the callback routine.

A driver removes notification registration by calling **IoUnregisterPlugPlayNotification** with the *NotificationEntry* returned by **IoRegisterPlugPlayNotification**.

Handling Hardware Profile Change Events

12/5/2018 • 2 minutes to read • Edit Online

At specific times during a hardware profile change, the PnP manager calls notification callback routines that registered for **EventCategoryHardwareProfileChange**:

- Before there is a change in the machine's hardware profile, the PnP manager calls registered notification callback routines and specifies a *NotificationStructure*.**Event** of GUID_HWPROFILE_QUERY_CHANGE.
- After the machine's hardware profile change is complete, the PnP manager calls registered notification callback routines and specifies a *NotificationStructure*.**Event** of GUID_HWPROFILE_CHANGE_COMPLETE.
- If the machine's hardware profile change is canceled, the PnP manager calls registered notification callback routines and specifies a *NotificationStructure*.**Event** of GUID_HWPROFILE_CHANGE_CANCELLED.

For a GUID_HWPROFILE_QUERY_CHANGE event the PnP manager calls user-mode callback routines and then calls kernel-mode callback routines. In response to a GUID_HWPROFILE_QUERY_CHANGE event, a driver's notification callback routine typically just returns STATUS_SUCCESS.

For a GUID_HWPROFILE_CHANGE_COMPLETE event the PnP manager calls kernel-mode callback routines and then calls user-mode callback routines. In response to such an event, a driver's callback routine might refresh its hardware-profile-specific settings.

For a GUID_HWPROFILE_CHANGE_CANCELLED event the PnP manager calls kernel-mode callback routines and then user-mode routines. In response to such an event, a driver's callback routine typically just returns STATUS_SUCCESS. If the driver performed any operations in response to the GUID_HWPROFILE_QUERY_CHANGE event, the driver would undo those operations in response to the cancellation event.

Using PnP Custom Notification

6/25/2019 • 2 minutes to read • Edit Online

A driver can use the target device change notification mechanism to be notified of custom events on a device.

The programmer that defines the custom event must do the following:

1. Define a new GUID for the custom event.

   Generate the GUID with **Uuidgen** or **Guidgen** (which are included in the Microsoft Windows SDK). Publish the GUID in an appropriate header file and documentation.

2. Write code to trigger the custom event.

   In kernel mode, a driver calls **IoReportTargetDeviceChange** with the custom GUID and a pointer to the PDO for the device. Custom events can only be triggered from kernel mode.

A driver writer uses custom notification with a procedure like the following:

1. The driver (or application) registers for notification of the custom event.

   In kernel mode, a driver calls **IoRegisterPlugPlayNotification** and registers for an **EventCategoryTargetDeviceChange** on the device.

   In user mode, an application registers using **RegisterDeviceNotification**. See the Windows SDK for further information.

2. A kernel-mode component triggers the custom event.

3. The PnP manager calls notification routines registered on the device.

   The PnP manager calls the registered user-mode callback routines and then calls the kernel-mode callback routines.

4. When user-mode notification is complete, the kernel-mode driver notification callback routine(s) respond to the custom event.

   See Guidelines for Writing PnP Notification Callback Routines for general guidelines for notification callback routines. In addition to those guidelines, a custom notification callback routine must not open a handle to a device from within the callback routine thread.

Introduction to Power Management

12/5/2018 • 2 minutes to read • Edit Online

Microsoft Windows supports a power management architecture that provides a comprehensive approach to system and device power management. This power management architecture is designed to meet ever-increasing user requirements, which include:

- Customers are demanding that their computers be automatically available at all times, even when turned off. For example, network administrators want to manage computers late at night, and home users want to use their computer to receive faxes. Users with computers hidden away under desks want to be able to turn them by pressing a button on the keyboard or monitor.
- Customers want to decrease the amount of power and total energy that a PC uses, whether the power comes from an electrical wall outlet or a battery.

To meet these ever-increasing user requirements, Windows must be able to manage the power that is used by any device in the system, including add-in boards such as graphics cards, network adapters, modems, and sound cards. To effectively manage power, the PC software, hardware, and Windows must work together in a framework that enables every device to be power managed in a consistent manner.

A PC configuration that takes full advantage of the Windows power management architecture, provides the following advantages to users:

- Energy savings and extended battery life.

  Reducing system power consumption results in lower energy costs and longer battery life.

- Minimal startup and shutdown delays.

  If a working state is not required, the system power state can be changed from the working state to a sleep state and, subsequently, quickly changed back to a working state as required. This allows the system to be responsive to the user, yet energy can be conserved during the time period that a working state is not required.

- Quiet operation.

  In many cases the full capabilities of a system might not be required. Powering down devices that are not being used can reduce noise. This capability is important in situations where near-silent operation is highly desirable, such as in Media Center PCs.

In systems that support power management, the computer and its peripheral devices are maintained at the lowest feasible power level to accomplish the tasks at hand. Drivers cooperate with the operating system to manage power for their devices. If all drivers support power management, the operating system can manage power consumption on a system-wide basis, thus conserving power, shutting down and resuming quickly, and waking up when required.

This integrated approach to power management—involving the operating system, system hardware, device drivers, and device hardware—results in the following:

- More intelligent power management decisions. At each level, the best-informed component directs power usage.
- Greater reliability. Better power management decisions reduce the chance of ill-timed shutdowns and loss of data.

- Platform independence. The operating system manages power in a controlled, uniform way across different hardware platforms, allowing maximum conservation of power on various devices.
- Better device integration. Device drivers that conform to industry-wide specifications ensure maximum conservation regardless of the hardware platform.

Combined, these advantages result in greater power conservation and more efficient usage than has previously been possible.

The industry-wide OnNow initiative defines the hardware and software requirements for power management. For more information, see Industry Initiatives for Power Management.

Industry Initiatives for Power Management

12/21/2018 • 2 minutes to read • Edit Online

The OnNow initiative defines hardware and software support required for power management.

The *Advanced Configuration and Power Interface Specification*, part of the OnNow initiative, defines a hardware-level interface that enables operating systems to implement power management in a consistent, platform-independent way.

A *Device Class Power Management Reference Specification* is available for each common device class, such as audio or communications devices. Each of these specifications defines power management requirements for a class of device. Driver writers should refer to these specifications, available through the ACPI / Power Management website, for device-specific details.

Support for Power Management

12/5/2018 • 2 minutes to read • Edit Online

To support power management, drivers must also support Plug and Play (PnP). Driver support for PnP is required because many power management operations are associated with installing and removing devices, and the PnP manager notifies drivers of these events by means of PnP IRPs. Additionally, drivers report device support for power management in response to PnP queries for device capabilities.

Power management works on two levels: one applies to individual devices and the other to the system as a whole.

The power manager, part of the operating system kernel, manages the power level of the entire system. If all drivers in the system support power management, the power manager can manage power consumption on a system-wide basis, utilizing not only the fully on and fully off states, but also various intermediate system sleep states.

Legacy drivers that were written before the operating system supported power management continue to work as they did previously. However, systems that include legacy drivers cannot enter any of the intermediate system sleep states; they can operate only in the fully on or fully off states as before.

Device power management applies to individual devices. A driver that supports power management can turn its device on when it is needed and off when it is not in use. Devices that have the hardware capability can enter intermediate device power states. The presence of legacy drivers in the system does not affect the ability of newer drivers to manage power for their devices.

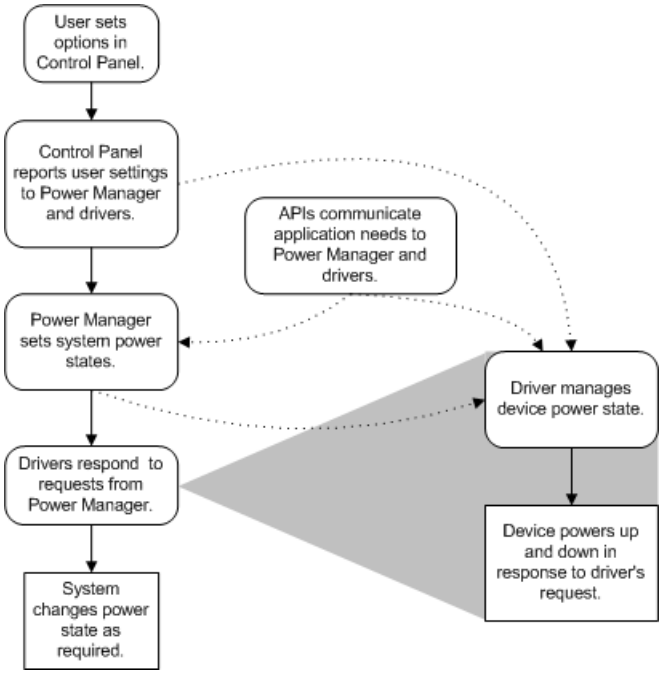
Beginning with Windows Vista, the operating system also supports driver performance states. Drivers that support device performance states can choose to tradeoff performance or features with a reduction in power consumption. Windows Vista provides a framework for devices to retrieve their power settings and information about the system power state. This mechanism is extendable, allowing driver vendors to define and install new custom power settings for their device. For more information, see System Power Policy.

System-Wide Overview of Power Management

6/25/2019 • 2 minutes to read • Edit Online

Power management requires support from system and device hardware and from system software and drivers. Required hardware support is covered in the industry specifications, as described in the previous section. This topic covers the software support—specifically, what drivers must do to conform to operating system requirements and to manage power as appropriate for their devices.

The following figure shows a system-wide overview of power management.



Applications and users can affect power management decisions through Control Panel and by calling power management routines. Users can use Control Panel to set system and device power options, including custom power settings. Control Panel notifies the power manager and drivers of changes to the active power policy and associated power settings. Beginning with Windows Vista, the power manager notifies a driver by calling the **power setting callback** that a driver registers to receive notifications. In Windows Server 2003, Windows XP, and Windows 2000, this notification is performed through WMI.

The power manager administers the system-wide *power policy*, the rules that govern the system's power usage. (For more information, see System Power Policy.) Using information from Control Panel and APIs, the power manager can determine when applications are using, or might need to use, various devices, so that it can adjust the system's power policy appropriately.

The power manager also provides an interface for drivers, comprising power management support routines, power management minor IRPs, and required driver entry points.

When the power manager requests a change to the system power state, drivers respond by putting their devices in an appropriate device power state. In addition, drivers can perform idle detection for their devices and put unused devices in a sleep state. Bus-specific mechanisms report device power capabilities, set and report device status, and change device power. Exactly how and when device power is changed depends on the type of device and the capabilities of the device hardware.

Although ACPI hardware realizes the greatest power savings, the hardware need not be ACPI-compliant for power management in drivers to be effective.

Power States

12/5/2018 • 2 minutes to read • Edit Online

A power state indicates the level of power consumption—and thus the extent of computing activity—by the system or by a single device. The power manager sets the power state of the system as a whole. Device drivers set the power state of their individual devices.

The ACPI specification defines two sets of discrete power states: *system power states* and *device power states*. Each power state has a unique name.

System power states are named S*x*, where *x* is a state number between 0 and 5. Device power states are named D*x*, where *x* is a state number between 0 and 3. The state number is inversely related to power consumption: higher numbered states use less power. States S0 and D0 are the highest-powered, most functional, fully on states. States S5 and D3 are the lowest-powered states and have the longest wake-up latency.

These clearly defined power states allow many devices from various manufacturers to work together consistently and predictably. For example, when the power manager sets the system in state S3, it can rely upon drivers that support power management not only to put their devices in the corresponding device power state but also to return to the working state in a predictable fashion.

ACPI BIOS

6/25/2019 • 2 minutes to read • Edit Online

The integrated power management features supported by Microsoft Windows operating systems are available only on computers that have an Advanced Configuration and Power Interface (ACPI) BIOS.

Windows Server 2003, Windows XP, and Windows 2000 require that an ACPI BIOS be dated January 1, 1999 or later. However, if one of these Windows versions determines that such a BIOS is known to exhibit ACPI problems, the loader disables ACPI and instead uses Advanced Power Management (APM). Beginning with Windows Vista, the operating system supports only a computer with an ACPI-compliant BIOS that is dated January 1, 1999 or later.

Device Manager shows whether an individual computer supports ACPI. Check the driver information for the **Computer** device category.

For more information about ACPI, see the ACPI 5.0 specification.

Acpi.sys: The Windows ACPI Driver

10/7/2019 • 4 minutes to read • Edit Online

The Windows ACPI driver, Acpi.sys, is an inbox component of the Windows operating system. The responsibilities of Acpi.sys include support for power management and Plug and Play (PnP) device enumeration. On hardware platforms that have an ACPI BIOS, the HAL causes Acpi.sys to be loaded during system startup at the base of the device tree. Acpi.sys acts as the interface between the operating system and the ACPI BIOS. Acpi.sys is transparent to the other drivers in the device tree.

Other tasks performed by Acpi.sys on a particular hardware platform might include reprogramming the resources for a COM port or enabling the USB controller for system wake-up.

In this topic

ACPI devices

The hardware platform vendor specifies a hierarchy of ACPI namespaces in the ACPI BIOS to describe the hardware topology of the platform. For more information, see ACPI Namespace Hierarchy.

For each device described in the ACPI namespace hierarchy, the Windows ACPI driver, Acpi.sys, creates either a filter device object (filter DO) or a physical device object (PDO). If the device is integrated into the system board, Acpi.sys creates a filter device object, representing an ACPI bus filter, and attaches it to the device stack immediately above the bus driver (PDO). For other devices described in the ACPI namespace but not on the system board, Acpi.sys creates the PDO. Acpi.sys provides power management and PnP features to the device stack by means of these device objects. For more information, see Device Stacks for an ACPI Device.

A device for which Acpi.sys creates a device object is called an ACPI device. The set of ACPI devices varies from one hardware platform to the next, and depends on the ACPI BIOS and the configuration of the motherboard. Note that Acpi.sys loads an ACPI bus filter only for a device that is described in the ACPI namespace and is permanently connected to the hardware platform (typically, this device is integrated into the core silicon or soldered to the system board). Not all motherboard devices have an ACPI bus filter.

All ACPI functionality is transparent to higher-level drivers. These drivers must make no assumptions about the presence or absence of an ACPI filter in any given device stack.

Acpi.sys and the ACPI BIOS support the basic functions of an ACPI device. To enhance the functionality of an ACPI device, the device vendor can supply a WDM function driver. For more information, see Operation of an ACPI Device Function Driver.

An ACPI device is specified by a definition block in the system description tables in the ACPI BIOS. A device's definition block specifies, among other things, an operation region, which is a contiguous block of device memory that is used to access device data. Only Acpi.sys modifies the data in an operation region. The device's function driver can read the data in an operation region but must not modify the data. When called, an operation region handler transfers bytes in the operation region to and from the data buffer in Acpi.sys. The combined operation of the function driver and Acpi.sys is device-specific and is defined in the ACPI BIOS by the hardware vendor. In general, the function driver and Acpi.sys access particular areas in an operation region to perform device-specific operations and retrieve information. For more information, see Supporting an Operation Region.

ACPI control methods

ACPI control methods are software objects that declare and define simple operations to query and configure ACPI devices. Control methods are stored in the ACPI BIOS and are encoded in a byte-code format called ACPI Machine Language (AML). The control methods for a device are loaded from the system firmware into the device's ACPI namespace in memory, and interpreted by the Windows ACPI driver, Acpi.sys.

To invoke a control method, the kernel-mode driver for an ACPI device initiates an **IRP_MJ_DEVICE_CONTROL** request, which is handled by Acpi.sys. For drivers loaded on ACPI-enumerated devices, Acpi.sys always implements the physical device object (PDO) in the driver stack. For more information, see Evaluating ACPI Control Methods.

ACPI specification

For the latest *Advanced Configuration and Power Interface Specification*, see the ACPI 5.0 specification available from the Unified Extensible Firmware Interface Forum website. Revision 5.0 of the ACPI specification introduces a set of features to support low-power, mobile PCs that are based on System on a Chip (SoC) integrated circuits and that implement the connected standby power model. Starting with Windows 8 and Windows 8.1, the Windows ACPI driver, Acpi.sys, supports the new features in the ACPI 5.0 specification. For more information, see Windows ACPI design guide for SoC platforms.

ACPI debugging

System integrators and ACPI device driver developers can use the Microsoft AMLI debugger to debug AML code. Because AML is an interpreted language, AML debugging requires special software tools. Checked versions of the Windows ACPI driver, Acpi.sys, contain a debugger component to support AML debugging. For more information about the AMLI debugger, see ACPI Debugging. For information about how to download a checked build of Windows, see Downloading a Checked Build of Windows. For information about compiling ACPI Source Language (ASL) into AML, see Microsoft ASL Compiler.

Power Manager

6/25/2019 • 2 minutes to read • Edit Online

The power manager is responsible for managing power usage for the system. It administers the system-wide power policy and tracks the path of power IRPs through the system.

The power manager requests power operations by sending **IRP_MJ_POWER** requests to drivers. A request can specify a new power state or can query whether a change in power state is feasible.

When sleep, hibernation, or shutdown is required, the power manager requests the appropriate power action by sending an **IRP_MJ_POWER** request to each leaf node in the device tree. The power manager considers the following in determining whether the system should sleep, hibernate, or shut down:

- System activity level
- System battery level
- Shutdown, hibernate, or sleep requests from applications
- User actions, such as pressing the power button
- Control panel settings

For more information, see Windows Kernel-Mode Power Manager.

Driver Role in Power Management

6/25/2019 • 2 minutes to read • Edit Online

Drivers support power management in two ways:

1. Drivers respond to system-wide power requests issued by the power manager.
2. Drivers manage power and performance states for their individual devices.

Every driver must have a *DispatchPower* routine to handle **IRP_MJ_POWER** requests. The *DispatchPower* routine must inspect each power IRP and either handle it or pass it down to the next-lower driver.

For a device to participate in power management, every driver in the device stack for the device must respond to or pass power IRPs appropriately. Failure of a single driver to act correctly can cause power management to be disabled across the entire system.

One driver for each device manages power policy for its device. That driver can send power IRPs to its own device stack to perform power operations on its device. The power policy manager is responsible for issuing device power IRPs that correspond to system power IRPs.

In addition, drivers might perform certain power tasks, such as powering on a device at start-up or powering off a device at removal, without receiving a power IRP. These are considered implicit power requests.

For more information, see Power Management Responsibilities for Drivers.

ACPI notifications

10/17/2019 • 9 minutes to read • Edit Online

Each ACPI notification that the PEP's *AcceptAcpiNotification* callback routine receives is accompanied by a Notification parameter that indicates the type of notification, and a Data parameter that points to a data structure that contains the information for the specified notification type.

In this call, the Notification parameter is set to a PEP_NOTIFY_ACPI_XXX constant value that indicates the notification type. The Data parameter points to a PEP_ACPI_XXX structure type that is associated with this notification type.

The following ACPI notification IDs are used by the AcceptAcpiNotification callback routine.

NOTIFICATION ID	VALUE	ASSOCIATED STRUCTURE
PEP_NOTIFY_ACPI_PREPARE_DEVICE	0x01	PEP_ACPI_PREPARE_DEVICE
PEP_NOTIFY_ACPI_ABANDON_DEVICE	0x02	PEP_ACPI_ABANDON_DEVICE
PEP_NOTIFY_ACPI_REGISTER_DEVICE	0x03	PEP_ACPI_REGISTER_DEVICE
PEP_NOTIFY_ACPI_UNREGISTER_DEVICE	0x04	PEP_ACPI_UNREGISTER_DEVICE
PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE	0x05	PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE
PEP_NOTIFY_ACPI_QUERY_OBJECT_INFORMATION	0x06	PEP_ACPI_QUERY_OBJECT_INFORMATION
PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD	0x07	PEP_ACPI_EVALUATE_CONTROL_METHOD
PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES	0x08	PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES
PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES	0x09	PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES

PEP_NOTIFY_ACPI_PREPARE_DEVICE

Notification: The value PEP_NOTIFY_ACPI_PREPARE_DEVICE. Data: A pointer to a PEP_ACPI_PREPARE_DEVICE structure that identifies the device by name.

Allows the PEP to choose whether to provide ACPI services for a device.

The Windows power management framework (PoFx) sends this notification when the Windows ACPI driver discovers a new device in the ACPI namespace during device enumeration. This notification is sent to PEPs that implement AcceptAcpiNotification callback routines.

To send a PEP_NOTIFY_ACPI_PREPARE_DEVICE notification, PoFx calls the PEP's AcceptAcpiNotification routine. In this call, the Notification parameter value is PEP_NOTIFY_ACPI_PREPARE_DEVICE, and the Data parameter

points to a PEP_ACPI_PREPARE_DEVICE structure that contains the name of the device. If the PEP is prepared to provide ACPI services for this device, the PEP sets the DeviceAccepted member of this structure to TRUE. To decline to provide such services, the PEP sets this member to FALSE.

If the PEP indicates (by setting DeviceAccepted = TRUE) that it is prepared to provide ACPI services for the device, PoFx will respond by sending the PEP a PEP_NOTIFY_ACPI_REGISTER_DEVICE notification to register the PEP to be the sole provider of ACPI services for the device. PoFx expects only one PEP to claim the role of ACPI services provider for a device.

As a best practice, do not perform any device initialization in response to the PEP_NOTIFY_ACPI_PREPARE_DEVICE notification. Instead, defer this initialization until either the PEP_NOTIFY_ACPI_REGISTER_DEVICE notification for the device is received, or an ACPI control method (for example, _INI) is invoked for the device.

For a PEP_NOTIFY_ACPI_PREPARE_DEVICE notification, the AcceptAcpiNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_ACPI_ABANDON_DEVICE

Notification: The value PEP_NOTIFY_ACPI_ABANDON_DEVICE.

Data: A pointer to a PEP_ACPI_ABANDON_DEVICE structure that identifies the abandoned device.

Informs the PEP that the specified device has been abandoned and no longer requires ACPI services from the PEP.

The Windows power management framework (PoFx) sends this notification to inform the PEP that the device is no longer in use by the operating system. The PEP can use this notification to clean up any internal storage that it has allocated to track the state of the device.

To send a PEP_NOTIFY_ACPI_ABANDON_DEVICE notification, PoFx calls the PEP's AcceptAcpiNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_ACPI_ABANDON_DEVICE, and the Data parameter points to a PEP_ACPI_ABANDON_DEVICE structure.

PoFx sends this notification only to a PEP that has opted to provide ACPI services for the device in a previous PEP_NOTIFY_ACPI_PREPARE_DEVICE notification. If the PEP has registered to provide these services in a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification, PoFx will send a PEP_NOTIFY_ACPI_UNREGISTER_DEVICE notification for the device before sending the PEP_NOTIFY_ACPI_ABANDON_DEVICE notification.

For a PEP_NOTIFY_ACPI_ABANDON_DEVICE notification, the AcceptAcpiNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_ACPI_REGISTER_DEVICE

Notification: The value PEP_NOTIFY_ACPI_REGISTER_DEVICE.

Data: A pointer to a PEP_ACPI_REGISTER_DEVICE structure that identifies the device. In response to this notification, the PEP is expected to create a valid PEPHANDLE value to identify the device and to write this handle value to the structure.

Registers the PEP to be the sole provider of ACPI services for the specified device.

The Windows power management framework (PoFx) sends this notification to a PEP that has indicated—in a previous PEP_NOTIFY_ACPI_PREPARE_DEVICE notification—that it is prepared to provide ACPI services for the specified device.

To send a PEP_NOTIFY_ACPI_REGISTER_DEVICE notification, PoFx calls the PEP's AcceptAcpiNotification routine. In this call, the Notification parameter value is PEP_NOTIFY_ACPI_REGISTER_DEVICE, and the Data

parameter points to a PEP_ACPI_REGISTER_DEVICE structure that identifies the device for which the PEP is to provide ACPI services.

For a PEP_NOTIFY_ACPI_REGISTER_DEVICE notification, the AcceptAcpiNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_ACPI_UNREGISTER_DEVICE

Notification: The value PEP_NOTIFY_ACPI_UNREGISTER_DEVICE.

Data: A pointer to a PEP_ACPI_UNREGISTER_DEVICE structure that contains the PEPHANDLE for the device.

Cancels the registration of the specified device for ACPI services from the PEP.

In response to this notification, the PEP can destroy the PEPHANDLE that the PEP created for this device in a previous PEP_NOTIFY_ACPI_REGISTER_DEVICE notification.

To send a PEP_NOTIFY_ACPI_UNREGISTER_DEVICE notification, PoFx calls the PEP's AcceptAcpiNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_ACPI_UNREGISTER_DEVICE, and the Data parameter points to a PEP_ACPI_UNREGISTER_DEVICE structure.

For a PEP_NOTIFY_ACPI_UNREGISTER_DEVICE notification, the AcceptAcpiNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE

Notification: The value PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE.

Data: A pointer to a PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure that contains an enumeration of the objects in the ACPI namespace of the device.

Queries the PEP for the list of ACPI objects (native methods) supported by the PEP under the specified device in the ACPI namespace.

The Windows ACPI driver uses the objects enumerated by this notification to build the namespace for the specified device. Thereafter, when referring to this device, the ACPI driver will query the PEP only for these objects.

The Windows power management framework (PoFx) sends the PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE notification shortly after a device is discovered and the PEP registers to provide ACPI services for the device. For more information about this registration, see PEP_NOTIFY_ACPI_REGISTER_DEVICE.

To send a PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE notification, PoFx calls the PEP's AcceptAcpiNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE, and the Data parameter points to a PEP_ACPI_ENUMERATE_DEVICE_NAMESPACE structure.

The AcceptAcpiNotification routine is expected to handle a PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE notification and to return TRUE. Failure to do so causes a bug check.

For a PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE notification, the AcceptAcpiNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_ACPI_QUERY_OBJECT_INFORMATION

Notification: The value PEP_NOTIFY_ACPI_QUERY_OBJECT_INFORMATION.

Data: A pointer to a PEP_ACPI_QUERY_OBJECT_INFORMATION structure that specifies the attributes of the

ACPI object.

Queries the PEP for information about a previously enumerated ACPI object.

The Windows power management framework (PoFx) sends this notification to query the PEP for the attributes of an object that was enumerated during the handling of a previous PEP_NOTIFY_ACPI_ENUMERATE_DEVICE_NAMESPACE notification. Currently, the only objects that are enumerated are control methods.

To send a PEP_NOTIFY_ACPI_QUERY_OBJECT_INFORMATION notification, PoFx calls the PEP's AcceptAcpiNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_ACPI_QUERY_OBJECT_INFORMATION, and the Data parameter points to a PEP_ACPI_QUERY_OBJECT_INFORMATION structure.

For a PEP_NOTIFY_ACPI_QUERY_OBJECT_INFORMATION notification, the AcceptAcpiNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD

Notification: The value PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD.

Data: A pointer to a PEP_ACPI_EVALUATE_CONTROL_METHOD structure that specifies an ACPI control method to evaluate, an input argument to supply to this method, and an output buffer for the result.

Is used to evaluate an ACPI control method for which the PEP is the registered handler.

The Windows power management framework (PoFx) sends this notification to the PEP when the Windows ACPI driver needs to evaluate an ACPI control method that is implemented by the PEP.

To send a PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification, PoFx calls the PEP's AcceptAcpiNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD, and the Data parameter points to a PEP_ACPI_EVALUATE_CONTROL_METHOD structure.

The platform designer can choose whether to have the PEP or the ACPI firmware handle a particular ACPI control method. If the PEP is the registered handler for an ACPI control method, PoFx responds to a request from the Windows ACPI driver to evaluate this method by sending a PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification to the PEP.

The following is a list of examples of ACPI control methods that the PEP can handle for a device:

Device identification and configuration: _HID, _CID, _UID, _ADR, _CLS, _SUB, _CRS, _PRS, and so on. Device power management and wake: _PS0 through _PS3, _PR0 through _PR3, _DSW, and so on. Device-specific methods: _DSM and any device-stack-specific control methods. For a special device, such as an ACPI Time and Alarm device, this notification is used to evaluate time and alarm methods (_GCP, _GRT, _SRT, and so on).

For a PEP_NOTIFY_ACPI_EVALUATE_CONTROL_METHOD notification, the AcceptAcpiNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES

Notification: The value PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES.

Data: A pointer to a PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES structure that contains the list of power resources.

Queries the PEP for the list of raw resources needed to control power to the device.

In response to this notification, the PEP provides the list of raw resources that are needed to control power to the

device. The Windows ACPI driver requires this list so that it can reserve the power resources required by the device, and provide the corresponding list of translated resources to the PEP (by sending a PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES notification). For more information, see Raw and Translated Resources.

To send a PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES notification, The Windows power management framework (PoFx) calls the PEP's AcceptAcpiNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES, and the Data parameter points to a PEP_ACPI_QUERY_DEVICE_CONTROL_RESOURCES structure.

For a PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES notification, the AcceptAcpiNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES

Notification: The value PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES.

Data: A pointer to a PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES structure that contains the list of translated resources.

Provides the PEP with a list of translated resources for any power control resources needed for the device.

The Windows power management framework (PoFx) sends this notification if the PEP listed any raw resources in response to the previous PEP_NOTIFY_ACPI_QUERY_DEVICE_CONTROL_RESOURCES notification. The PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES notification provides the PEP with the corresponding list of translated resources. For more information, see Raw and Translated Resources.

To send a PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES notification, PoFx calls the PEP's AcceptAcpiNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES, and the Data parameter points to a PEP_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES structure.

For a PEP_NOTIFY_ACPI_TRANSLATED_DEVICE_CONTROL_RESOURCES notification, the AcceptAcpiNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_ACPI_WORK

Notification: The value PEP_NOTIFY_ACPI_WORK.

Data: A pointer to a PEP_WORK structure.

Sent to the PEP once each time the PEP calls the RequestWorker routine to request an item of work from the Windows power management framework (PoFx). This notification is used for ACPI-only work.

After the PEP calls the RequestWorker routine to request a work item, PoFx responds by sending the PEP a PEP_NOTIFY_ACPI_WORK notification. However, this notification is not sent until the resources (that is, the worker thread) necessary to process the work item are available. In this way, PoFx guarantees that the work request that the PEP passes to PoFx during the notification can never fail due to lack of resources.

On entry, the PEP should assume that the PEP_WORK structure is uninitialized. To handle this notification, the PEP should set the WorkInformation member to point to a PEP-allocated PEP_WORK_INFORMATION structure that describes the work that is being requested. In addition, the PEP should set the NeedWork member of the PEP_WORK structure to TRUE to confirm that the PEP has handled the PEP_NOTIFY_ACPI_WORK notification and that the WorkInformation member points to a valid PEP_WORK_INFORMATION structure. If the PEP fails to handle the notification or is unable to allocate the PEP_WORK_INFORMATION structure, the PEP should set the WorkInformation member to NULL and set the NeedWork member to FALSE.

For a PEP_NOTIFY_ACPI_WORK notification, the AcceptAcpiNotification routine is always called at IRQL =

PASSIVE_LEVEL.

Device power management (DPM) notifications

12/5/2018 • 26 minutes to read • Edit Online

Each device power management (DPM) notification that the PEP's AcceptDeviceNotification callback routine receives is accompanied by a Notification parameter that indicates the type of notification, and a Data parameter that points to a data structure that contains the information for the specified notification type.

In this call, the Notification parameter is set to a PEP_DPM_XXX constant value that indicates the notification type. The Data parameter points to a PEP_XXX structure type that is associated with this notification type.

The following DPM notification IDs are used by the AcceptDeviceNotification callback routine.

PEP_DPM_PREPARE_DEVICE

Notification The value PEP_DPM_PREPARE_DEVICE.

Data A pointer to a PEP_PREPARE_DEVICE structure. Tells the PEP that owns the specified device to configure the device to operate in the D0 (working) device power state.

The Windows power management framework (PoFx) sends this notification to the PEP before a device's driver stack is started for the first time by the operating system. This notification allows the PEP to turn on any external power or clock resources that are required to operate the device.

To send a PEP_DPM_PREPARE_DEVICE notification, the operating system calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_PREPARE_DEVICE, and the Data parameter points to a PEP_PREPARE_DEVICE structure. On entry, the DeviceId member of this structure is a device identification string that uniquely identifies a device. Before returning, the PEP sets the DeviceAccepted member of this structure to TRUE to claim ownership of the device, or to FALSE to indicate that it does not own the device.

The PEP that owns the power management for a device is responsible for managing power and clock resources that are external to the device and that are needed to operate the device. This PEP enables the clock signal and power to the device in response to a PEP_DPM_PREPARE_DEVICE notification, and removes the clock signal and power from the device in response to a PEP_DPM_ABANDON_DEVICE notification.

The following table shows the preconditions that are in effect when this operating system sends a PEP_DPM_PREPARE_DEVICE notification to the PEP, and the postconditions that must be in effect after the PEP handles this notification for a device that it owns.

Preconditions The device can be in any power state. Postconditions If the PEP claims ownership of the device, the device and all its components must be turned on, and clocks to the device must ungated. The PEP can receive PEP_DPM_PREPARE_DEVICE notifications for multiple devices as the power manager tries to find PEP owners for these devices. The PEP should set the DeviceAccepted member of the PEP_PREPARE_DEVICE structure to FALSE for all devices that the PEP does not own.

No PEP_DPM_PREPARE_DEVICE notifications are sent for core devices.

For a PEP_DPM_PREPARE_DEVICE notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_ABANDON_DEVICE

Notification The value PEP_DPM_ABANDON_DEVICE.

Data A pointer to a PEP_ABANDON_DEVICE structure. Tells the PEP that the specified device is no longer being used by the operating system.

The Windows power management framework (PoFx) sends this notification to the PEP after the operating system removes a device's driver stack. This notification allows the PEP to turn off any external power or clock resources that are used to operate the device, and to remove this device from future decision-making processes. If the device must be started again later, the PEP will first receive a PEP_DPM_PREPARE_DEVICE notification.

To send a PEP_DPM_ABANDON_DEVICE notification, the operating system calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_ABANDON_DEVICE, and the Data parameter points to a PEP_ABANDON_DEVICE structure. On entry, the DeviceId member of this structure is a device identification string that uniquely identifies a device. Before returning, the PEP sets the DeviceAccepted member of this structure to TRUE to claim ownership of the device, or to FALSE to indicate that it does not own the device.

The PEP that owns the power management for a device is responsible for managing power and clock resources that are external to the device and that are needed to operate the device.

The following table shows the preconditions that are in effect when this operating system sends a PEP_DPM_ABANDON_DEVICE notification to the PEP, and the postconditions that must be in effect after the PEP handles this notification for a device that it owns.

Preconditions The PEP has received a PEP_DPM_PREPARE_DEVICE notification for the device and accepted ownership of the device. If the PEP has received a PEP_DPM_REGISTER_DEVICE notification for the device and accepted the device registration, it has subsequently received a PEP_DPM_UNREGISTER_DEVICE notification for the device. Postconditions Any resources that were allocated in response to the PEP_DPM_PREPARE_DEVICE notification must be freed. For a PEP_DPM_PREPARE_DEVICE notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_REGISTER_DEVICE

Notification The value PEP_DPM_REGISTER_DEVICE.

Data A pointer to a PEP_REGISTER_DEVICE_V2 structure. Tells the PEP that the driver stack for the specified device has registered with the Windows power management framework (PoFx).

PoFx sends this notification when the device's driver stack calls the PoFxRegisterDevice routine to register the device. This notification allows the PEP to copy the device's registration information to the PEP's internal storage for later reference.

To send a PEP_DPM_REGISTER_DEVICE notification, the operating system calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_REGISTER_DEVICE, and the Data parameter points to a PEP_REGISTER_DEVICE_V2 structure that contains the device's kernel handle and other registration information. On entry, the DeviceId member of this structure is a device identification string that uniquely identifies a device. Before returning, the PEP sets the DeviceAccepted member of this structure to TRUE to claim ownership of the device, or to FALSE to indicate that it does not own the device. For information about the other members of this structure, see PEP_REGISTER_DEVICE_V2.

The following table shows the preconditions that are in effect when this operating system sends a PEP_DPM_REGISTER_DEVICE notification to the PEP, and the postconditions that must be in effect after the PEP handles this notification for a device that it owns.

Condition type Description Preconditions The PEP has received a PEP_DPM_PREPARE_DEVICE notification for a device that it owns. Postconditions The PEP is ready to receive other device power management (DPM) notifications associated with this device.

For a PEP_DPM_REGISTER_DEVICE notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_UNREGISTER_DEVICE

Notification The value PEP_DPM_UNREGISTER_DEVICE.

Data A pointer to a PEP_UNREGISTER_DEVICE structure. Tells the PEP that owns the specified device that the device's driver stack has withdrawn its registration from the Windows power management framework (PoFx).

PoFx sends this notification to inform the PEP that any registration information that the PEP stored for the device during the previous PEP_DPM_REGISTER_DEVICE notification is no longer valid. In response, the PEP can clean up any internal state used for power management of this device.

To send a PEP_DPM_UNREGISTER_DEVICE notification, the operating system calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_UNREGISTER_DEVICE, and the Data parameter points to a PEP_UNREGISTER_DEVICE structure. This structure contains the handle that the PEP created in response to the previous PEP_DPM_REGISTER_DEVICE notification for the device.

The following table shows the preconditions that are in effect when this operating system sends a PEP_DPM_UNREGISTER_DEVICE notification to the PEP, and the postconditions that must be in effect after the PEP handles this notification for a device that it owns.

Preconditions If the PEP has received a PEP_DPM_REGISTER_DEVICE notification for the device and accepted device registration. The PEP can receive any device power management (DPM) notifications associated with this device. The PEP can report "work" associated with this device. Postconditions The PEP can no longer receive any device power management (DPM) notifications associated with this device, except for PEP_DPM_ABANDON_DEVICE. The PEP cannot report "work" associated with this device. For a PEP_DPM_UNREGISTER_DEVICE notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_DEVICE_POWER_STATE

Notification The value PEP_DPM_DEVICE_POWER_STATE.

Data A pointer to a PEP_DEVICE_POWER_STATE structure. Sent to the PEP each time the device's driver stack either requests a change to a new Dx power state, or a previously requested transition to a Dx power state completes.

After the PEP calls the RequestWorker routine to request a work item, PoFx responds by sending the PEP a PEP_DPM_DEVICE_POWER_STATE notification. However, this notification is not sent until the resources (that is, the worker thread) necessary to process the work item are available. In this way, PoFx guarantees that the work request that the PEP passes to PoFx during the notification can never fail due to lack of resources.

To send a PEP_DPM_DEVICE_POWER_STATE notification, the operating system calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_DEVICE_POWER_STATE, and the Data parameter points to a PEP_DEVICE_POWER_STATE structure. On entry, the PEP should assume that the contents of this structure are uninitialized. To handle this notification, the PEP should set the WorkInformation member to point to a PEP-allocated PEP_WORK_INFORMATION structure that describes the work that is being requested. In addition, the PEP should set the NeedWork member of the PEP_WORK structure to TRUE to confirm that the PEP has handled the PEP_DEVICE_POWER_STATE notification and that the WorkInformation member points to a valid PEP_WORK_INFORMATION structure. If the PEP fails to handle the notification or is unable to allocate the PEP_WORK_INFORMATION structure, the PEP should set the WorkInformation member to NULL and set the NeedWork member to FALSE.

For a PEP_DPM_DEVICE_POWER_STATE notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_COMPONENT_ACTIVE

Notification The value PEP_DPM_COMPONENT_ACTIVE.

Data A pointer to a PEP_COMPONENT_ACTIVE structure that identifies the component and that indicates whether this component is making a transition to the active condition or to the idle condition. Informs the PEP that a component needs to make a transition from the idle condition to the active condition, or vice versa.

The Windows power management framework (PoFx) sends this notification when a transition is pending either to the active condition or to the idle condition.

To send a PEP_DPM_COMPONENT_ACTIVE notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_COMPONENT_ACTIVE, and the Data parameter points to a PEP_COMPONENT_ACTIVE structure.

A component that is accessible is in the active condition. A component that is inaccessible is in the idle condition. A component that is in the active condition is always in the F0 component power state. The component cannot leave F0 until it enters the idle condition. A component that is in the idle condition might be in F0 or in a low-power Fx state. The active/idle condition of a component is the only reliable means for a driver to determine whether a component is accessible. A component that is in F0 but is also in the idle condition might be about to switch to a low-power Fx state.

When an active component is ready to enter the idle condition, the transition occurs immediately. During the handling of the PEP_DPM_COMPONENT_ACTIVE notification, the PEP might, for example, request a transition from F0 to a low-power Fx state for the component.

If a component is in a low-power Fx state when a PEP_DPM_COMPONENT_ACTIVE notification requests a transition from the idle condition to the active condition, the PEP must first switch the component to F0 before the component can enter the active condition. The PEP might need to finish preparing the component for the transition to the active condition asynchronously, after returning from the AcceptDeviceNotification callback for the PEP_DPM_COMPONENT_ACTIVE notification. After the component is fully configured to operate in the active condition, the PEP must call the RequestWorker routine and then handle the resulting PEP_DPM_WORK notification by setting WorkType = PepWorkActiveComplete in the PEP_WORK_INFORMATION structure.

If the PEP receives a PEP_DPM_COMPONENT_ACTIVE notification for a component that is in F0 and is already fully configured to operate in the active condition, the PEP might be able to finish handling this notification synchronously. If "fast path" handling of the notification is supported, the WorkInformation member of the PEP_COMPONENT_ACTIVE structure for this notification contains a pointer to a PEP_WORK_INFORMATION structure, and the PEP can set the WorkType member of this structure to PepWorkActiveComplete to complete the transition. However, if WorkInformation = NULL, no "fast path" is available and the PEP must complete the transition asynchronously by calling RequestWorker, as described in the preceding paragraph.

For more information about the active and idle conditions, see Component-Level Power Management.

For a PEP_DPM_COMPONENT_ACTIVE notification, the AcceptDeviceNotification routine is called at IRQL <= DISPATCH_LEVEL.

PEP_DPM_WORK

Notification The value PEP_DPM_WORK.

Data A pointer to a PEP_WORK structure. Sent to the PEP once each time the PEP calls the RequestWorker routine to request an item of work from the Windows power management framework (PoFx).

After the PEP calls the RequestWorker routine to request a work item, PoFx responds by sending the PEP a PEP_DPM_WORK notification. However, this notification is not sent until the resources (that is, the worker thread) necessary to process the work item are available. In this way, PoFx guarantees that the work request that the PEP passes to PoFx during the notification can never fail due to lack of resources.

To send a PEP_DPM_WORK notification, the operating system calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_WORK, and the Data parameter points to a PEP_WORK structure. On entry, the PEP should assume that the contents of this structure are uninitialized. To handle this notification, the PEP should set the WorkInformation member to point to a PEP-allocated PEP_WORK_INFORMATION structure that describes the work that is being requested. In addition, the PEP should set the NeedWork member of the PEP_WORK structure to TRUE to confirm that the PEP has handled the PEP_DPM_WORK notification and that the WorkInformation member points to a valid PEP_WORK_INFORMATION structure. If the PEP fails to handle the notification or is unable to allocate the PEP_WORK_INFORMATION structure, the PEP should set the WorkInformation member to NULL and set the NeedWork member to FALSE.

For a PEP_DPM_WORK notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_POWER_CONTROL_REQUEST

Notification The value PEP_DPM_POWER_CONTROL_REQUEST.

Data A pointer to a PEP_POWER_CONTROL_REQUEST structure. Informs the PEP that a driver has called the PoFxPowerControl API to send a control code directly to the PEP.

The Windows power management framework (PoFx) sends this notification to the PEP when a driver calls the PoFxPowerControl API to send a control code directly to the PEP. The notification Data pointer in this case points to the PEP_POWER_CONTROL_REQUEST structure.

Power control requests and their semantics are defined between the PEP writer and the device class owner. Typically such an interface is for device class specific communication that is not captured in the generalized power management framework. For example, the UART controller may communicate baud rate information to the PEP to modify some platform clock rails / dividers and such communication would likely leverage a power control request.

Note The PEP can only request to send a control code to the device after it receives either a PEP_DPM_DEVICE_STARTED notification or PEP_DPM_POWER_CONTROL_REQUEST notification.

For a PEP_DPM_POWER_CONTROL_REQUEST notification, the AcceptDeviceNotification routine is called at IRQL <= DISPATCH_LEVEL.

PEP_DPM_POWER_CONTROL_COMPLETE

Notification The value PEP_DPM_POWER_CONTROL_COMPLETE.

Data A pointer to a PEP_POWER_CONTROL_COMPLETE structure. Informs the PEP that a driver has completed a power control request that was previously issued by the PEP

The Windows power management framework (PoFx) sends this notification to the PEP when a driver completes a power control request issued previously by the PEP.

Note The PEP can ignore this notification if it does not issue any power control requests.

For a PEP_DPM_POWER_CONTROL_COMPLETE notification, the AcceptDeviceNotification routine is called at IRQL <= DISPATCH_LEVEL.

PEP_DPM_SYSTEM_LATENCY_UPDATE

Notification The value PEP_DPM_SYSTEM_LATENCY_UPDATE.

Data A pointer to a PEP_SYSTEM_LATENCY structure. Informs the PEP that the OS has updated the overall system latency tolerance.

The Windows power management framework (PoFx) sends this notification when the OS updates the overall system latency tolerance.

In earlier versions of PoFx, this notification was used by the PEP for processor and platform idle state selection. With the latest PEP interfaces, the selection process is entirely handled by the OS and as such this notification is no longer useful. It is included here for completeness and the PEP should ignore it.

To send a PEP_DPM_SYSTEM_LATENCY_UPDATE notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. For this notification, the AcceptDeviceNotification routine is called at IRQL <= DISPATCH_LEVEL.

PEP_DPM_DEVICE_STARTED

Notification The value PEP_DPM_DEVICE_STARTED.

Data A pointer to a PEP_DEVICE_STARTED structure. Informs the PEP that the device has started so that it is available to receive power control transactions.

Device stacks register with the OS for runtime power management in a two-step process. The driver first calls PoFxRegisterDevice to provide information about the number of components, their idle states and corresponding attributes. In response to this call, the PEP receives a PEP_DPM_REGISTER_DEVICE notification.

After registration succeeds, the driver has the opportunity to initialize its components (i.e. set active, update latency requirements, update expected idle residency, etc.). Once the driver has completed any initialization tasks, it notifies the power manager by calling PoFxStartDevicePowerManagement. In response, the PEP will receive a PEP_DPM_DEVICE_STARTED notification. At this point, the device is considered to be fully enabled for runtime power management.

As a result, the PEP cannot issue any power control requests to the driver unless it has either first received a PEP_DPM_DEVICE_STARTED notification or a PEP_DPM_POWER_CONTROL_REQUEST notification.

Note The PEP can ignore this notification if it does not issue any power control requests.

For a PEP_DPM_DEVICE_STARTED notification, the AcceptDeviceNotification routine is called at IRQL <= DISPATCH_LEVEL.

PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE

Notification The value PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE.

Data A pointer to a PEP_NOTIFY_COMPONENT_IDLE_STATE structure. Sent to the PEP when the OS issues an idle state transition for a given component.

The Windows power management framework (PoFx) sends this notification when the OS issues an idle state transition for a given component.

Important The PEP must handle this notification.

For each idle state transition, the PEP is notified before and after the driver is notified. The PEP distinguishes between pre and post notifications by examining the DriverNotified member of the PEP_NOTIFY_COMPONENT_IDLE_STATE structure. For a post-notification, the DriverNotified member will be TRUE.

Pre-notifications are generally used when transitioning to F0. In this case the PEP may need to re-enable clock or power resources such that when the driver handles the F0 notification, the hardware is available. Accordingly, post-

notifications are generally used when transitioning from F0 to a deeper idle state. After a driver has handled the idle state notification, the PEP can safely turn off clock and power resources.

Handling an idle state transition for a given component may require asynchronous processing if the operation takes a significant amount of time or the IRQL is too high to complete the transition synchronously. As a result, the PEP can complete this notification synchronously or asynchronously by setting the Completed member to TRUE or FALSE respectively.

If the notification is to be completed asynchronously, the PEP notifies the OS on completion by requesting a worker (see RequestWorker) and filling out the provided work information structure in the resulting PEP_DPM_WORK notification using a work type of PepWorkCompleteIdleState.

To send a PEP_DPM_NOTIFY_COMPONENT_IDLE_STATE notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. This routine is called at IRQL <= DISPATCH_LEVEL.

PEP_DPM_REGISTER_DEBUGGER

Notification The value PEP_DPM_REGISTER_DEBUGGER.

Data A pointer to a PEP_REGISTER_DEBUGGER structure. Informs the PEP that a registered device may be used as a debug port.

The Windows power management framework (PoFx) sends this notification to notify the PEP that a registered device may be used as a debug port.

For a PEP_DPM_REGISTER_DEBUGGER notification, the AcceptDeviceNotification routine is called at IRQL <= DISPATCH_LEVEL.

PEP_DPM_REGISTER_CRASHDUMP_DEVICE

Notification The value PEP_DPM_REGISTER_CRASHDUMP_DEVICE.

Data A pointer to a PEP_REGISTER_CRASHDUMP_DEVICE structure. The Windows power management framework (PoFx) sends this notification when a device registers as a crashdump handler.

The ability to generate a memory dump (crashdump) when the system encounters a fatal error is invaluable toward determining the cause of the crash. Windows, by default, will generate a crashdump when the system encounters a bugcheck. In this context, the system is under a very constrained operating environment with interrupts disabled and the system IRQL at HIGH_LEVEL.

Since devices involved in writing a crashdump to disk (i.e. storage controller, PCI controller, etc.) may be powered down at the time of the crash, the OS must call into the PEP to power on the device. As such, the OS requests a callback (PowerOnDumpDeviceCallback) from the PEP for every device on the crashdump stack and invokes the callback when generating the dump file.

Given the constrained environment at the time of the crash, the callback provided by the PEP must not access paged code, block on any events or invoke any code that may do the same. Furthermore, the process of powering up any required resources cannot rely on interrupts. As a result, the PEP may have to revert to polling should it need to wait for various resources to be enabled. If the PEP cannot power on the device under these constraints, it should either not handle the notification or not supply a callback routine.

To send a PEP_DPM_REGISTER_CRASHDUMP_DEVICE notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. For this notification, the AcceptDeviceNotification routine is called at IRQL <= HIGH_LEVEL.

PEP_DPM_DEVICE_IDLE_CONSTRAINTS

Notification The value PEP_DPM_DEVICE_IDLE_CONSTRAINTS.

Data A pointer to a PEP_DEVICE_PLATFORM_CONSTRAINTS structure. Sent to the PEP to query for dependencies between device D-states and platform idle states.

The Windows power management framework (PoFx) sends this notification to the PEP to query for dependencies between device D-states and platform idle states. The PEP uses this notification to return the lightest D-state the device can still be in and enter each platform idle state. The OS will guarantee the device is in the minimum D-state before entering an associated platform idle state. If a platform idle state does not depend on this device being in any D-state, the PEP should specify a minimum D-state of PowerDeviceD0. If no platform idle states depend on this device being in a particular D-state, this notification can be ignored.

This notification is sent to each device after the PEP has received the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification.

To send a PEP_DPM_DEVICE_IDLE_CONSTRAINTS notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_DEVICE_IDLE_CONSTRAINTS, and the Data parameter points to a PEP_DEVICE_PLATFORM_CONSTRAINTS structure.

For a PEP_DPM_DEVICE_IDLE_CONSTRAINTS notification, the AcceptDeviceNotification routine is always called at IRQL = DISPATCH_LEVEL.

PEP_DPM_COMPONENT_IDLE_CONSTRAINTS

Notification The value PEP_DPM_COMPONENT_IDLE_CONSTRAINTS.

Data A pointer to a PEP_COMPONENT_PLATFORM_CONSTRAINTS structure. Sent to the PEP to query for dependencies between component F-states and platform idle states.

The Windows power management framework (PoFx) sends this notification to the PEP to query for dependencies between component F-states and platform idle states. The PEP uses this notification to return the lightest F-state the component can still be in and enter each platform idle state. The OS will guarantee the component is in the minimum F-state before entering an associated platform idle state. If a platform idle state does not depend on this component being in any F-state, the PEP should specify a minimum F-state of 0. If no platform idle states depend on this component being in a particular F-state, this notification can be ignored.

Device idle constraints deeper than D0 are more constraining than component idle states for components on the device. For a given platform idle state index, if the device specified a device idle constraint, the corresponding component idle constraint for all components associated with the device are ignored.

This notification is sent to each component on each device after the PEP receives a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification.

To send a PEP_DPM_COMPONENT_IDLE_CONSTRAINTS notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. The AcceptDeviceNotification routine is always called at IRQL = DISPATCH_LEVEL.

PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES

Notification The value PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES.

Data A pointer to a PEP_QUERY_COMPONENT_PERF_CAPABILITIES structure. Informs the PEP that it is being queried for the number of performance state (P-state) sets that are defined for a component.

To send a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES, and the Data parameter points to a PEP_QUERY_COMPONENT_PERF_CAPABILITIES structure.

For a PEP_DPM_QUERY_COMPONENT_PERF_CAPABILITIES notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_QUERY_COMPONENT_PERF_SET

Notification The value PEP_DPM_QUERY_COMPONENT_PERF_SET.

Data A pointer to a PEP_QUERY_COMPONENT_PERF_SET structure. Informs the PEP that it is being queried for information about a set of performance state values (P-state set) for a component.

To send a PEP_DPM_QUERY_COMPONENT_PERF_SET notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_QUERY_COMPONENT_PERF_SET, and the Data parameter points to a PEP_QUERY_COMPONENT_PERF_SET structure.

For a PEP_DPM_QUERY_COMPONENT_PERF_SET notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_QUERY_COMPONENT_PERF_SET_NAME

Notification The value PEP_DPM_QUERY_COMPONENT_PERF_SET_NAME.

Data A pointer to a PEP_QUERY_COMPONENT_PERF_SET_NAME structure. Informs the PEP that it is being queried for information about a set of performance state values (P-state set) for a component.

To send a PEP_DPM_QUERY_COMPONENT_PERF_SET_NAME notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_QUERY_COMPONENT_PERF_SET_NAME, and the Data parameter points to a PEP_QUERY_COMPONENT_PERF_SET_NAME structure.

For a PEP_DPM_QUERY_COMPONENT_PERF_SET_NAME notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_QUERY_COMPONENT_PERF_STATES

Notification The value PEP_DPM_QUERY_COMPONENT_PERF_STATES.

Data A pointer to a PEP_QUERY_COMPONENT_PERF_STATES structure. Informs the PEP that it is being queried for a list of discrete performance state (P-state) values for a specified P-state set.

To send a PEP_DPM_QUERY_COMPONENT_PERF_STATES notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_QUERY_COMPONENT_PERF_STATES, and the Data parameter points to a PEP_QUERY_COMPONENT_PERF_STATES structure.

For a PEP_DPM_QUERY_COMPONENT_PERF_STATES notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_REGISTER_COMPONENT_PERF_STATES

Notification The value PEP_DPM_REGISTER_COMPONENT_PERF_STATES.

Data A pointer to a PEP_REGISTER_COMPONENT_PERF_STATES structure. Informs the PEP about the performance states (P-states) of the specified component.

To send a PEP_DPM_REGISTER_COMPONENT_PERF_STATES notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_REGISTER_COMPONENT_PERF_STATES, and the Data parameter points to a

PEP_REGISTER_COMPONENT_PERF_STATES structure.

For a PEP_DPM_REGISTER_COMPONENT_PERF_STATES notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_REQUEST_COMPONENT_PERF_STATE

Notification The value PEP_DPM_REQUEST_COMPONENT_PERF_STATE.

Data A pointer to a PEP_REQUEST_COMPONENT_PERF_STATE structure. Informs the PEP that one or more performance state (P-state) changes are requested by the Windows power management framework (PoFx).

To send a PEP_DPM_REQUEST_COMPONENT_PERF_STATE notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_REQUEST_COMPONENT_PERF_STATE, and the Data parameter points to a PEP_REQUEST_COMPONENT_PERF_STATE structure.

For a PEP_DPM_REQUEST_COMPONENT_PERF_STATE notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_QUERY_CURRENT_COMPONENT_PERF_STATE

Notification The value PEP_DPM_QUERY_CURRENT_COMPONENT_PERF_STATE.

Data A pointer to a PEP_QUERY_CURRENT_COMPONENT_PERF_STATE structure. Informs the PEP that it is being queried for information about the current P-state in the specified P-state set.

To send a PEP_DPM_QUERY_CURRENT_COMPONENT_PERF_STATE notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. In this call, the Notification parameter value is PEP_DPM_QUERY_CURRENT_COMPONENT_PERF_STATE, and the Data parameter points to a PEP_QUERY_CURRENT_COMPONENT_PERF_STATE structure.

For a PEP_DPM_QUERY_CURRENT_COMPONENT_PERF_STATE notification, the AcceptDeviceNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_DPM_QUERY_DEBUGGER_TRANSITION_REQUIREMENTS

Notification The value PEP_DPM_QUERY_DEBUGGER_TRANSITION_REQUIREMENTS.

Data A pointer to a PEP_DEBUGGER_TRANSITION_REQUIREMENTS structure. Sent to the PEP to query for the set of coordinated or platform states which require the debugger to be powered off.

The Windows power management framework (PoFx) sends this notification to the PEP to query for the set of coordinated or platform states which require the debugger to be powered off. If this notification is accepted, the OS will perform all debugger power transitions for the PEP, and the PEP may not use TransitionCriticalResource to power manage the debugger.

This notification is sent to each debugger device after the PEP has accepted a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE or PEP_NOTIFY_PPM_QUERY_COORDINATED_STATES notification.

To send a PEP_DPM_QUERY_DEBUGGER_TRANSITION_REQUIREMENTS notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. For this notification, the AcceptDeviceNotification routine is always called at IRQL = DISPATCH_LEVEL.

PEP_DPM_LOW_POWER_EPOCH

Notification The value PEP_DPM_LOW_POWER_EPOCH.

Data A pointer to a PEP_LOW_POWER_EPOCH structure. This notification is deprecated.

PEP_DPM_QUERY_SOC_SUBSYSTEM

Notification The value PEP_DPM_QUERY_SOC_SUBSYSTEM.

Data A pointer to a PEP_QUERY_SOC_SUBSYSTEM structure. Sent to the PEP to collect basic information about a particular system on a chip (SoC) subsystem.

The Windows power management framework (PoFx) sends this notification to the PEP after platform idle states have been initialized in order to collect basic information about a particular SoC subsystem. A PEP that does not implement SoC subsystem accounting, or does not implement it for the specified platform idle state, returns FALSE. This directs the OS to stop sending diagnostic notifications to the PEP for this platform idle state.

A system's SubsystemCount and a subsystem's MetadataCount can change with PEP/BSP updates. SubsystemIndex can change every time the OS boots.

Important The PEP cannot ignore this notification. The PEP is receiving this notification because it responded to the PEP_DPM_QUERY_SOC_SUBSYSTEM_COUNT notification for this PlatformIdleStateIndex with a non-zero SubsystemCount.

To send a PEP_DPM_QUERY_SOC_SUBSYSTEM notification, PoFx calls the PEP's AcceptDeviceNotification callback routine at IRQL < DISPATCH_LEVEL.

PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME

Notification The value PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME.

Data A pointer to a PEP_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME structure. Sent to the PEP when the OS wants to collect the tally of time a particular system on a chip (SoC) subsystem has blocked entry into a particular platform idle state without the OS's knowledge.

Typically the OS calls this notification at the end of an extended connected standby session where the OS attempted to enter the specified platform idle state. The PEP_QUERY_SOC_SUBSYSTEM_COUNT.SubsystemCount value, filled in earlier by the PEP during subcomponent initialization, specifies how many PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notifications are sent to the PEP at a time. A PEP can receive multiple PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notifications for a given subsystem. These notifications may or may not be interleaved with PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notifications.

Important The PEP cannot ignore this notification. The PEP is receiving this notification because it responded to the PEP_DPM_QUERY_SOC_SUBSYSTEM_COUNT notification for this PlatformIdleStateIndex with a non-zero SubsystemCount.

To send a PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notification, PoFx calls the PEP's AcceptDeviceNotification callback routine at IRQL < DISPATCH_LEVEL.

PEP_DPM_QUERY_SOC_SUBSYSTEM_COUNT

Notification The value PEP_DPM_QUERY_SOC_SUBSYSTEM_COUNT.

Data A pointer to a PEP_QUERY_SOC_SUBSYSTEM_COUNT structure. Sent to the PEP after platform idle states have been initialized to tell the OS whether the PEP supports system on a chip (SoC) subsystem accounting for a given platform idle state.

This is the first SoC subsystem diagnostic notification sent to the PEP. A PEP that does not implement SoC subsystem accounting, or does not implement it for the specified platform idle state, returns FALSE, in which case

the OS will not send the PEP any more SoC subsystem diagnostic notifications for this platform idle state.

Note The PEP can ignore this notification if it does not implement SoC diagnostic notifications for the specified platform idle state.

To send a PEP_DPM_QUERY_SOC_SUBSYSTEM_COUNT notification, PoFx calls the PEP's AcceptDeviceNotification callback routine at IRQL < DISPATCH_LEVEL.

PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA

Notification The value PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA.

Data A pointer to a PEP_QUERY_SOC_SUBSYSTEM_METADATA structure. Sent to the PEP to collect optional metadata about the subsystem whose blocking time has just been queried.

This notification is typically sent to the PEP immediately following a PEP_DPM_QUERY_SOC_SUBSYSTEM_BLOCKING_TIME notification. One PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification collects all key-value metadata pairs describing the subsystem.

Important The PEP cannot ignore this notification. The PEP is receiving this notification because it responded to the PEP_DPM_QUERY_SOC_SUBSYSTEM_COUNT notification for this PlatformIdleStateIndex with a non-zero SubsystemCount.

To send a PEP_DPM_QUERY_SOC_SUBSYSTEM_METADATA notification, PoFx calls the PEP's AcceptDeviceNotification callback routine. For this notification, the AcceptDeviceNotification routine is called at IRQL < DISPATCH_LEVEL.

PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING

Notification The value PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING.

Data A pointer to a A pointer to a PEP_RESET_SOC_SUBSYSTEM_ACCOUNTING structure. structure. Sent to the PEP to clear all subsystem blocking time and metadata accounting, perform any additional initialization required, and restart the accounting.

The Windows power management framework (PoFx) sends this notification to the PEP anytime after all subsystems are initialized with the OS. Typically, this notification is called when the OS begins a new period of analysis around what is keeping the system on a chip (SoC) out of the specified platform idle state (targeting DRIPS upon entering connected standby). The OS only sends this notification for platform idle states for which the PEP initialized one or more SoC subsystems.

To send a PEP_DPM_RESET_SOC_SUBSYSTEM_ACCOUNTING notification, PoFx calls the PEP's AcceptDeviceNotification callback routine at IRQL < DISPATCH_LEVEL.

Processor power management (PPM) notifications

10/16/2019 • 30 minutes to read • Edit Online

Each processor power management (PPM) notification that the PEP's *AcceptProcessorNotification* callback routine receives is accompanied by a Notification parameter that indicates the type of notification, and a Data parameter that points to a data structure that contains the information for the specified notification type.

In this call, the Notification parameter is set to a PEP_NOTIFY_PPM_XXX constant value that indicates the notification type. The Data parameter points to a PEP_PPM_XXX structure type that is associated with this notification type.

The following processor power management (PPM) notification IDs are used by the AcceptProcessorNotification callback routine.

PEP_NOTIFY_PPM_QUERY_CAPABILITIES

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_CAPABILITIES.

Data

A pointer to a PEP_PPM_QUERY_CAPABILITIES structure.

Remarks

Informs the PEP that it is being queried for the power management capabilities of the PEP.

The Windows power management framework (PoFx) sends this notification when the PEP is queried for its power management capabilities. This happens at processor initialization time and will be sent for each processor in the system.

Platforms with x86/AMD64 processors must use ACPI interfaces for processor performance control.

To send a PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification, PoFx calls the PEP's AcceptProcessorNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_PPM_QUERY_CAPABILITIES, and the Data parameter points to a PEP_PPM_QUERY_CAPABILITIES structure.

For a PEP_NOTIFY_PPM_QUERY_CAPABILITIES notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_QUERY_IDLE_STATES

Notification

The value PEP_NOTIFY_PPM_QUERY_IDLE_STATES.

Data

A pointer to a PEP_PPM_QUERY_IDLE_STATES structure.

Remarks

Informs the PEP about idle states.

To send a PEP_NOTIFY_PPM_QUERY_IDLE_STATES notification, PoFx calls the PEP's AcceptProcessorNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_PPM_QUERY_IDLE_STATES, and the Data parameter points to a PEP_PPM_QUERY_IDLE_STATES structure.

For a PEP_NOTIFY_PPM_QUERY_IDLE_STATES notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_IDLE_SELECT

Notification

The value PEP_NOTIFY_PPM_IDLE_SELECT.

Data

A pointer to a PEP_PPM_IDLE_SELECT structure.

Remarks

Informs the PEP of idle select.

To send a PEP_NOTIFY_PPM_IDLE_SELECT notification, PoFx calls the PEP's AcceptProcessorNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_PPM_IDLE_SELECT, and the Data parameter points to a PEP_PPM_IDLE_SELECT structure.

For a PEP_NOTIFY_PPM_IDLE_SELECT notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_IDLE_CANCEL

Notification

The value PEP_NOTIFY_PPM_IDLE_CANCEL.

Data

A pointer to a PEP_PPM_IDLE_CANCEL structure.

Remarks

Informs the PEP of a cancel action.

To send a PEP_NOTIFY_PPM_IDLE_CANCEL notification, PoFx calls the PEP's AcceptProcessorNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_PPM_IDLE_CANCEL, and the Data parameter points to a PEP_PPM_IDLE_CANCEL structure.

For a PEP_NOTIFY_PPM_IDLE_CANCEL notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_IDLE_EXECUTE

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_IDLE_EXECUTE.

Data

A pointer to a PEP_PPM_IDLE_EXECUTE or PEP_PPM_IDLE_EXECUTE_V2 structure.

Remarks

Sent to the PEP to transition the current processor to the specified idle state.

The Windows power management framework (PoFx) sends this notification to the PEP to transition the current processor to the specified idle state.

The PEP can prepare the hardware to enter the previously selected idle state, including notifying the OS of core system resources which may be affected by the sleep transition. Then it must execute the halt instruction to transition the processor to the idle state. Upon return from the idle state, the PEP must undo the hardware setup, including notifying the OS of core system resources which may have become active upon wake. If the PEP is unable to execute the processor (and platform) idle state, then it must return back an error status.

When using the coordinated idle state interface, the OS uses the PEP_PPM_IDLE_EXECUTE_V2 structure which includes the CoordinatedStateCount and CoordinatedState fields with the list of coordinated idle states that will be entered by the idle transition. The PlatformState field will specify the deepest platform coordinated idle state that's being entered, if any.

When not using the coordinated idle state interface, the OS uses the PEP_PPM_IDLE_EXECUTE structure.

For a PEP_NOTIFY_PPM_IDLE_EXECUTE notification, the AcceptProcessorNotification routine is called with interrupts disabled.

PEP_NOTIFY_PPM_IDLE_COMPLETE

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor.

Notification

The value PEP_NOTIFY_PPM_IDLE_COMPLETE.

Data

A pointer to a PEP_PPM_IDLE_COMPLETE or PEP_PPM_IDLE_COMPLETE_V2 structure..

Remarks

Informs the PEP that the current processor is waking up from a completed idle transition.

The Windows power management framework (PoFx) sends this notification when the current processor is waking up from a completed idle transition. If the platform was executing a platform idle transition, the first processor to wake will indicate the platform idle state that is being exited. Depending on the type of synchronization used in the platform idle transition, the first processor to wake from a platform idle state may not be the processor that entered the platform idle state.

If the processor was executing a deep idle state, the PEP must not wait until it receives the complete notification to restore core context or notify the OS that core resources have been restored. The OS expects these resources to have been restored once the execute notification has completed. When the hypervisor is enabled, the PEP will only receive this notification upon exit from a platform idle state, and with the ProcessorState field set to PEP_PROCESSOR_IDLE_STATE_UNKNOWN.

When using the coordinated idle state interface, the OS uses the PEP_PPM_IDLE_COMPLETE_V2 structure which includes the CoordinatedStateCount and CoordinatedState fields with the list of coordinated idle states that will be exited by the idle transition. The PlatformState field will specify the deepest platform coordinated idle state that's being exited, if any. Note that the set of coordinated idle states exited by this processor may be different from the set of coordinated idle states entered by it, if loose synchronization is used.

When not using the coordinated idle state interface, the OS uses the PEP_PPM_IDLE_COMPLETE structure.

For a PEP_NOTIFY_PPM_IDLE_COMPLETE notification, the AcceptProcessorNotification routine is called with interrupts disabled and is always executed on the target processor.

PEP_NOTIFY_PPM_IS_PROCESSOR_HALTED

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_IS_PROCESSOR_HALTED.

Data

A pointer to a PEP_PPM_IS_PROCESSOR_HALTED structure.

Remarks

Sent to the PEP to determine if the specified processor is currently halted in its selected idle state.

The Windows power management framework (PoFx) sends this notification to the PEP to determine if the specified processor is currently halted in its selected idle state. The OS will use this notification to check if a secondary processor has fully completed the transition to idle when coordinating platform idle states. The PEP must guarantee the target processor has reached a state in which the platform idle transition can safely occur (e.g., by checking hardware registers to see if the core is halted). Once this notification indicates the processor is in an idle state, that processor must not wake up unless the OS explicitly requests it to do so.

The PEP may receive this notification any time between the IDLE_SELECT and IDLE_COMPLETE notifications. It is guaranteed to receive this notification at most once during an idle transition.

For a PEP_NOTIFY_PPM_IS_PROCESSOR_HALTED notification, the AcceptProcessorNotification routine is called at any IRQL and with interrupts disabled, at any IRQL, and is never executed on the target processor.

<= HIGH_LEVEL

PEP_NOTIFY_PPM_INITIATE_WAKE

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor.

Notification

The value PEP_NOTIFY_PPM_INITIATE_WAKE.

Data

A pointer to a structure.

Remarks

Sent to the PEP for a specified processor to initiate its wake up from a non-interruptible idle state.

The Windows power management framework (PoFx) sends this notification to the PEP for a specified processor to initiate its wake up from a non-interruptible idle state. The PEP must return the status of wake for the target processor using NeedInterruptForCompletion. It returns TRUE if the processor requires an interrupt to finish waking up from the idle state. In this case the PEP must ensure the target processor is interruptible upon return from handling this notification. If the target processor is already running and/or will eventually exit the idle state (and is in the process of doing so) without requiring any software generated interrupt, NeedInterruptForCompletion should be set to FALSE.

Note The PEP will not receive this notification for the same processor concurrently.

For a PEP_NOTIFY_PPM_INITIATE_WAKE notification, the AcceptProcessorNotification routine is called at any IRQL, with interrupts disabled, and is never executed on the target processor.

<= HIGH_LEVEL

PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS.

Data

A pointer to a PEP_PPM_QUERY_FEEDBACK_COUNTERS structure.

Remarks

Informs the PEP that the PEP is being queried for the list of feedback counters that it supports.

The Windows power management framework (PoFx) sends this notification at processor initialization to query the PEP for the list of feedback counters that it supports.

For a PEP_NOTIFY_PPM_QUERY_FEEDBACK_COUNTERS notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_FEEDBACK_READ

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_FEEDBACK_READ.

Data

A pointer to a PEP_PPM_FEEDBACK_READ structure.

Remarks

Informs the PEP that it is being queried for a feedback counter's current value.

The Windows power management framework (PoFx) sends this notification when it wants to query a feedback

counter's current value.

This notification may be sent with interrupts disabled. If the counter's Affinitized field is set, this notification is executed on the target processor. Otherwise, this notification may be executed from any processor.

For a PEP_NOTIFY_PPM_FEEDBACK_READ notification, the AcceptProcessorNotification routine may be called at IRQL = DISPATCH_LEVEL.

PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES.

Data

A pointer to a PEP_PPM_QUERY_PERF_CAPABILITIES structure.

Remarks

Informs the PEP that it is being queried for the performance ranges supported by the platform.

The Windows power management framework (PoFx) sends this notification at processor initialization to query the performance ranges supported by the platform. The DomainId and DomainMembers fields of the PEP_PPM_QUERY_PERF_CAPABILITIES structure are used to express performance state domains to the platform. Each processor is a member of exactly one performance state domain. The operating system will ensure that all processors in a performance domain change performance together.

For a PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_PERF_CONSTRAINTS

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor.

Notification

The value PEP_NOTIFY_PPM_PERF_CONSTRAINTS.

Data

A pointer to a PEP_PPM_PERF_CONSTRAINTS structure.

Remarks

Informs the PEP that it is being queried for the current operating constraints of the processor.

The Windows power management framework (PoFx) sends this notification when it wants to inspect the current operating constraints of the processor. The PEP initiates a request for the OS to re-evaluate the perf constraints of the processor by executing a power control with the control code GUID_PPM_PERF_CONSTRAINT_CHANGE. The InBuffer and OutBuffer must be NULL.

The PEP must wait until it receives a PEP_DPM_DEVICE_STARTED notification for a processor before it issues a power control transaction for the processor.

For a PEP_NOTIFY_PPM_PERF_CONSTRAINTS notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_PERF_SET

This notification informs the PEP that the current operating performance of the processor should be changed.

The following describe parameters to *AcceptProcessorNotification*.

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value **PEP_NOTIFY_PPM_PERF_SET**.

Data

A pointer to a **PEP_PPM_PERF_SET** structure.

Remarks

The Windows power management framework (PoFx) sends this notification when it wants to change the current operating performance of the processor. This notification may be sent while executing on any processor.

For a PEP_NOTIFY_PPM_PERF_SET notification, the *AcceptProcessorNotification* routine is always called at IRQL = DISPATCH_LEVEL.

PEP_NOTIFY_PPM_PARK_SELECTION

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_PARK_SELECTION.

Data

A pointer to a PEP_PPM_PARK_SELECTION structure.

Remarks

Informs the PEP that the OS would like it to select a preferred set of processor cores to park.

The Windows power management framework (PoFx) sends this notification to instruct the PEP to select a preferred set of cores to park.

The PEP_NOTIFY_PPM_PARK_SELECTION has been overloaded to perform two functions:

Let the PEP select which processors (from the set of all processors in the system) should be parked, and which should be unparked. Let the PEP select which processors (from the set of all processors that are unparked) should receive interrupts, and which should not receive interrupts. Windows does not provide a means for the PEP to distinguish which of the two the OS is performing. As a result, when the PEP receives this notification with a given set of inputs (AdditionalUnparkedProcessors count and PoPreference), it should provide a consistent output (PepPreference) unless some external event causes a change in PEP preference.

For a PEP_NOTIFY_PPM_PARK_SELECTION notification, the AcceptProcessorNotification routine is always called

at IRQL = DISPATCH_LEVEL.

PEP_NOTIFY_PPM_CST_STATES

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_CST_STATES.

Data

A pointer to a PEP_PPM_CST_STATES structure.

Remarks

Sent to the PEP to indicate the set of ACPI-defined C-states supported by the processor.

The Windows power management framework (PoFx) sends this notification to the PEP to indicate the set of ACPI-defined C-states supported by the processor. This notification will be sent once before the first time the PEP receives PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification for a processor, and again any time that processor receives a Notify(0x81) indicating the _CST object has changed.

For a PEP_NOTIFY_PPM_CST_STATES notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES.

Data

A pointer to a PEP_PPM_QUERY_PLATFORM_STATES structure.

Remarks

Sent at processor initialization to query the number of platform idle states that the PEP supports.

The Windows power management framework (PoFx) sends this notification to the PEP at processor initialization to query the number of platform idle states that it supports. This notification is sent once upon boot. After returning a non-zero number of platform states, the PEP can then begin to select platform idle states during processor idle transitions.

For a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_QUERY_LP_SETTINGS

Notification

The value PEP_NOTIFY_PPM_QUERY_LP_SETTINGS.

Data

A pointer to a PEP_PPM_QUERY_LP_SETTINGS structure.

Remarks

To send a PEP_NOTIFY_PPM_QUERY_LP_SETTINGS notification, PoFx calls the PEP's AcceptProcessorNotification callback routine. In this call, the Notification parameter value is PEP_NOTIFY_PPM_QUERY_LP_SETTINGS, and the Data parameter points to a PEP_PPM_QUERY_LP_SETTINGS structure.

For a PEP_NOTIFY_PPM_QUERY_LP_SETTINGS notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2.

Data

A pointer to a PEP_PPM_QUERY_IDLE_STATES_V2 structure.

Remarks

Used at processor initialization to query the list of idle states that the PEP supports.

The Windows power management framework (PoFx) sends this notification to the PEP at processor initialization to query the list of idle states that it supports.

The Count member specifies the size of the idle state array. The processor driver will query for the number of idle states with PEP_NOTIFY_PPM_QUERY_CAPABILITIES before sending this notification.

The PEP fills in the IdleStates array with information about each idle state that it supports. The idle states should be listed in order of decreasing power consumption/increasing transition cost. The PEP is not required to report the same list of idle states for each processor.

For a PEP_NOTIFY_PPM_QUERY_IDLE_STATES_V2 notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE.

Data

A pointer to a PEP_PPM_QUERY_PLATFORM_STATE structure.

Remarks

Sent to the PEP to query the properties of a single platform idle state.

The Windows power management framework (PoFx) sends this notification at processor initialization to query the properties of a single platform idle state.

The StateIndex parameter of the PEP_PPM_QUERY_PLATFORM_STATE structure specifies the index of the platform idle state being queried. The processor driver will query for the number of supported platform idle states with PEP_NOTIFY_PPM_QUERY_PLATFORM_STATES before sending this notification. The processor driver will then send one PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification for each platform idle state. The processor driver will wait to send this notification until after all processors have registered with the PEP.

The PEP fills in State structure with information about the platform idle state. Platform idle states should be listed in order of decreasing power consumption/increasing transition cost.

For a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_TEST_IDLE_STATE

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_TEST_IDLE_STATE.

Data

A pointer to a PEP_PPM_TEST_IDLE_STATE structure.

Remarks

Used to test whether the specified processor and platform idle state can be entered on the specified processor.

The Windows power management framework (PoFx) sends this notification to test whether the specified processor and platform idle state can be entered on the specified processor. If the idle state can be entered, the PEP fills in veto code PEP_IDLE_VETO_NONE and completes the idle transition. If the idle transition cannot be completed for some reason, the PEP fills in a non-zero veto code.

Important Veto codes in the range 0x80000000 to 0xffffffff are reserved for OS use and may not be used.

When this notification is sent, the OS has already validated that all constraints associated with the selected processor or platform idle state have been met, including device and processor constraints for a platform idle transition.

This notification will be sent before the OS attempts to enter any processor or platform idle state, except for the processor idle state with index 0, which must always be enterable. Completing this notification with PEP_IDLE_VETO_NONE does not guarantee that the OS will enter the indicated idle state. This notification is sent with interrupts disabled. This notification is always executed on the target processor.

For a PEP_NOTIFY_PPM_TEST_IDLE_STATE notification, the AcceptProcessorNotification routine is called with interrupts disabled.

PEP_NOTIFY_PPM_IDLE_PRE_EXECUTE

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_IDLE_PRE_EXECUTE.

Data

A pointer to a PEP_PPM_IDLE_EXECUTE or PEP_PPM_IDLE_EXECUTE_V2 structure.

Remarks

Sent to the PEP to prepare the system to transition to the specified idle state.

The Windows power management framework (PoFx) sends this notification to the PEP to prepare the system to transition to the specified idle state. Upon successful completion of this notification, the OS will transition the processor into idle by entering the associated C-state. If the PEP is unable to prepare the system to enter the processor (and platform) idle state, then it must return back an error status.

When the hypervisor is enabled, the PEP will only receive this notification upon entry to a platform idle state, and with the ProcessorState field set to PEP_PROCESSOR_IDLE_STATE_UNKNOWN.

When using the coordinated idle state interface, the OS uses the PEP_PPM_IDLE_EXECUTE_V2 structure which includes the CoordinatedStateCount and CoordinatedState fields with the list of coordinated idle states that will be entered by the idle transition. The PlatformState field will specify the deepest platform coordinated idle state that's being entered, if any.

When not using the coordinated idle state interface, the OS uses the PEP_PPM_IDLE_EXECUTE structure.

For a PEP_NOTIFY_PPM_IDLE_PRE_EXECUTE notification, the AcceptProcessorNotification routine is called with interrupts disabled and is always executed on the target processor.

PEP_NOTIFY_PPM_UPDATE_PLATFORM_STATE

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_UPDATE_PLATFORM_STATE.

Data

A pointer to a PEP_PPM_QUERY_PLATFORM_STATE structure.

Remarks

Informs the PEP that a processor has received Notify(0x81) to update the characteristics of a platform idle state.

The Windows power management framework (PoFx) sends this notification when a processor has received Notify(0x81) to update the characteristics of a platform idle state. This notification is sent once for each platform idle state. If the PEP does not accept the notification (i.e. returns FALSE from its AcceptProcessorNotification callback), then the prior definition of the platform idle state, from the most recently accepted PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE or PEP_NOTIFY_PPM_UPDATE_PLATFORM_STATE notification, is preserved.

This notification uses the same Data buffer as the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification.

For a PEP_NOTIFY_PPM_UPDATE_PLATFORM_STATE notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE_RESIDENCIES

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE_RESIDENCIES.

Data

A pointer to a PEP_PPM_PLATFORM_STATE_RESIDENCIES structure.

Remarks

Informs the PEP that it should capture the actual accumulated time spent in each platform idle state since boot.

The Windows power management framework (PoFx) sends this notification to the PEP to capture the actual accumulated time spent in each platform idle state since boot. As such, this query is only applicable to platforms where the underlying hardware may autonomously decide to enter a platform idle state different from that requested by the OS. The values returned are used for diagnostic purposes and identify when the OS's view of platform idle state residency differs significantly from what the platform actually achieved.

Count specifies the number of elements in the States array, where the element index corresponds to platform idle state index. The PEP will fill each element with the matching state's actual residency and transition count.

Note The accumulated values captured by this query should correspond only to those periods where the PEP (or processor driver) actually executed a platform idle state transition. This will ensure that the comparison between OS calculated residency and actual residency is meaningful.

For a PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE_RESIDENCIES notification, the AcceptProcessorNotification routine can be called at any IRQL.

PEP_NOTIFY_PPM_QUERY_VETO_REASONS

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_VETO_REASONS.

Data

A pointer to a PEP_PPM_QUERY_VETO_REASONS structure.

Remarks

Used to query the number of unique veto reasons that the PEP uses in the ProcessorIdleVeto and PlatformIdleVeto callbacks.

The Windows power management framework (PoFx) sends this notification at processor initialization to query the number of unique veto reasons that the PEP uses in the ProcessorIdleVeto and PlatformIdleVeto callbacks. This notification is optional, and may be ignored by the PEP.

If accepted, the PEP is allowed to use the veto reasons between 1 and VetoReasonCount, inclusive, to veto any processor, platform, or coordinated idle state. The PEP is not allowed to use veto reasons greater than VetoReasonCount. The OS will pre-allocate veto tracking structures, and when used with PEP_NOTIFY_PPM_ENUMERATE_BOOT_VETOES, guarantees that all processor, platform, and coordinated state veto callbacks will succeed.

If this notification is not accepted by the PEP, the PEP may use the ProcessorIdleVeto and PlatformIdleVeto callbacks with any legal veto reason. The OS does not guarantee that the callbacks will not fail due to allocation failures or other issues.

For a PEP_NOTIFY_PPM_QUERY_VETO_REASONS notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_QUERY_VETO_REASON

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_VETO_REASON.

Data

A pointer to a PEP_PPM_QUERY_VETO_REASON structure.

Remarks

Sent to the PEP to query for information about a specific veto reason.

The Windows power management framework (PoFx) sends this notification at processor initialization to query for information about a specific veto reason. This notification is sent twice for each veto reason, once with a NULLName buffer to retrieve the allocation size needed for Name, and once with a non-NULLName buffer to fill in the contents of Name. The name should be a human-readable string indicating what condition this veto reason represents. Debugging tools such as WPA and the kernel debugger will display Name when diagnosing why an idle state was not entered.

For a PEP_NOTIFY_PPM_QUERY_VETO_REASON notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_ENUMERATE_BOOT_VETOES

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_ENUMERATE_BOOT_VETOES.

Data

The NULL pointer value.

Remarks

Informs the PEP that the OS is ready to accept calls to ProcessorIdleVeto or PlatformIdleVeto.

The Windows power management framework (PoFx) sends this notification after processor initialization but before first idle entry to indicate that the OS is ready to accept calls to ProcessorIdleVeto or PlatformIdleVeto. The PEP may enumerate any boot-time vetoes in the context of this notification, and the OS guarantees that they will take effect before the first attempt to select a processor, platform, or coordinated idle state. This notification has no associated Data parameter.

For a PEP_NOTIFY_PPM_ENUMERATE_BOOT_VETOES notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_PARK_MASK

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_PARK_MASK.

Data

A pointer to a PEP_PPM_PARK_MASK structure.

Remarks

Informs the PEP of the current core parking mask.

The Windows power management framework (PoFx) sends this notification at runtime to inform the PEP of the current core parking mask.

For a PEP_NOTIFY_PPM_PARK_MASK notification, the AcceptProcessorNotification routine is called at IRQL = DISPATCH_LEVEL and may be sent while executing on any processor.

PEP_NOTIFY_PPM_PARK_SELECTION_V2

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_PARK_SELECTION_V2.

Data

A pointer to a PEP_PPM_PARK_SELECTION_V2 structure.

Remarks

Informs the PEP that the OS would like it to select a preferred set of cores to park or steer interrupts away from. If this notification is not accepted, the OS will fall back to sending the PEP_NOTIFY_PPM_PARK_SELECTION notification.

When running its performance check algorithm, the OS may send the PEP_NOTIFY_PPM_PARK_SELECTION_V2 notification multiple times: zero or more times for each core efficiency class within each park domain, and zero or more times for interrupt steering. To assist the PEP in providing a consistent response to the OS for a performance check, the OS will supply the interrupt time based timestamp of the performance check evaluation that prompted the notification. All park selection notifications resulting from one performance check evaluation will have the same

timestamp. Note that the remaining fields (Count, AdditionalUnparkedProcessors, EvaluationType, and Processors) may vary for notifications that are sent during the same performance check evaluation, the PEP cannot assume that they will remain the same.

For a PEP_NOTIFY_PPM_PARK_SELECTION notification, the AcceptProcessorNotification routine is always called at IRQL = DISPATCH_LEVEL.

PEP_NOTIFY_PPM_PERF_CHECK_COMPLETE

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_PERF_CHECK_COMPLETE.

Data

A pointer to a PEP_PPM_PERF_CHECK_COMPLETE structure.

Remarks

Informs the PEP that the periodic performance check evaluation has completed.

The Windows power management framework (PoFx) sends this notification at runtime to notify the PEP that the periodic per check evaluation has completed.

For a PEP_NOTIFY_PPM_PERF_CHECK_COMPLETE notification, the AcceptProcessorNotification routine is called at IRQL = DISPATCH_LEVEL and may be sent while executing on any processor.

PEP_NOTIFY_PPM_QUERY_COORDINATED_DEPENDENCY

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_COORDINATED_DEPENDENCY.

Data

A pointer to a PEP_PPM_QUERY_COORDINATED_DEPENDENCY structure.

Remarks

Sent to the PEP to query for the dependencies of each coordinated idle state.

The Windows power management framework (PoFx) sends this notification at processor initialization to query the PEP for the dependencies of each coordinated idle state. The OS will allocate MaximumDependencySize elements for the Dependencies array. The PEP must fill in the number of elements of the array that were used in DependencySizeUsed.

If the dependency being expressed is on a processor, PEP fills in the TargetProcessor field with the POHANDLE of the target processor. The ExpectedState field then refers to the index of a processor idle state on the target processor.

If the dependency being expressed is on other coordinated idle states, PEP fills in NULL for the TargetProcessor.

The ExpectedState field then refers to the index of a coordinated idle state.

Each dependency lists a menu of options the OS is allowed to use to satisfy the dependency. When going idle, the OS will attempt to satisfy the dependency by checking the conditions for each, from highest index to lowest index. If the conditions for a dependency are met, then the OS will consider the dependency met. If none of the conditions can be met, the dependency is not met and the coordinated idle state may not be entered.

For a PEP_NOTIFY_PPM_QUERY_COORDINATED_DEPENDENCY notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_QUERY_COORDINATED_STATE_NAME

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_COORDINATED_STATE_NAME.

Data

A pointer to a PEP_PPM_QUERY_STATE_NAME structure.

Remarks

Sent to the PEP to query for information about a specific coordinated or platform idle state.

The Windows power management framework (PoFx) sends this notification at processor initialization to query the PEP for information about a specific coordinated or platform idle state. This notification is sent twice for each state, once with a NULL Name buffer to retrieve the allocation size needed for Name, and once with a non-NULL Name buffer to fill in the contents of Name. The name should be a human-readable string indicating the name of the coordinated idle state. Coordinated idle states should have unique names, except on multi-cluster systems, where the names of equivalent states on different clusters may be the same. Debugging tools such as WPA and the kernel debugger will display Name in diagnostics that refer to this coordinated/platform idle state.

For a PEP_NOTIFY_PPM_QUERY_COORDINATED_STATE_NAME notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_QUERY_COORDINATED_STATES

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_COORDINATED_STATES.

Data

A pointer to a PEP_PPM_QUERY_COORDINATED_STATES structure.

Remarks

Used at processor initialization to query for the properties of all coordinated idle states.

The Windows power management framework (PoFx) sends this notification to the PEP at processor initialization to query for the properties of all coordinated idle states. This notification is sent just before the PEP would have sent

the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification. If accepted, the PEP is using the coordinated idle state interface and will not receive any PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notifications. If not accepted, the PEP is using the platform idle state interface and the OS will fall back to using the PEP_NOTIFY_PPM_QUERY_PLATFORM_STATE notification to query for coordinated idle states.

The OS will wait to send this notification until after all processors have registered with the PEP.

The PEP fills in the State structure with information about the coordinated idle states.

The order of coordinated idle states must follow the following rules:

Two coordinated states that represent different power states for the same functional unit should be listed in order from lightest (most power consumption/least transition cost) to deepest (least power consumption/most transition cost). Coordinated idle states may only depend on other coordinated idle states with a lower index. There is not required order between two disjoint coordinated idle states (that is, two coordinated idle states that depend on disjoint sets of processors).

For a PEP_NOTIFY_PPM_QUERY_COORDINATED_STATES notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_QUERY_PROCESSOR_STATE_NAME

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_PROCESSOR_STATE_NAME.

Data

A pointer to a PEP_PPM_QUERY_STATE_NAME structure.

Remarks

Sent to the PEP to query for information about a specific processor idle state.

The Windows power management framework (PoFx) sends this notification at processor initialization to query the PEP for information about a specific processor idle state. This notification is sent twice for each state, once with a NULL Name buffer to retrieve the allocation size needed for Name, and once with a non-NULL Name buffer to fill in the contents of Name. The name should be a human-readable string indicating the name of the coordinated idle state. Coordinated idle states should have unique names, except on multi-cluster systems, where the names of equivalent states on different clusters may be the same. Debugging tools such as WPA and the kernel debugger will display Name in diagnostics that refer to this coordinated/platform idle state.

For a PEP_NOTIFY_PPM_QUERY_PROCESSOR_STATE_NAME notification, the AcceptProcessorNotification routine is always called at IRQL = PASSIVE_LEVEL.

PEP_NOTIFY_PPM_ENTER_SYSTEM_STATE

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_ENTER_SYSTEM_STATE.

Data

A pointer to a PEP_PPM_ENTER_SYSTEM_STATE structure.

Remarks

PEP_NOTIFY_PPM_ENTER_SYSTEM_STATE is an optional notification that notifies the PEP that the system is about to enter a system power state. This notification is sent to all processors simultaneously after the system has completed all passive level work transitioning the processor to the system power state.

This notification is sent at DISPATCH_LEVEL, with all processors at dispatch. This notification is always executed on the target processor.

Note The PEP must not queue any work from this notification. The processors will not process work items, DPCs, etc. after this notification has been sent.

DISPATCH_LEVEL

PEP_NOTIFY_PPM_PERF_SET_STATE

The following describe parameters to *AcceptProcessorNotification*.

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value **PEP_NOTIFY_PPM_PERF_SET_STATE**.

Data

A pointer to a **PEP_PPM_PERF_SET_STATE** structure.

Remarks

Used at runtime to set the current operating performance state of the processor. If the PEP has autonomous hardware capable of boosting/reducing performance without a performance set request, it should limit the requests from autonomous hardware based on the minimum performance state and/or maximum performance state, and target the desired performance state. Otherwise, it should run at exactly the desired performance state.

This notification is sent at DISPATCH_LEVEL. If scheduler directed performance states are in use, the PEP must adhere to the restrictions in section 3.3.6 when processing this notification. It may be sent while executing on any processor.

PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_DISCRETE_PERF_STATES.

Data

A pointer to a PEP_PPM_QUERY_DISCRETE_PERF_STATES structure. Used at processor initialization to query for the list of discrete performance states that the PEP supports, if the PEP_NOTIFY_PPM_QUERY_CAPABILITIES

notification indicates support for discrete performance states.

The performance state list should be ordered from fastest to slowest, with each performance state mapping to a distinct performance value. The performance state list should also include an entry that matches each performance value listed in the PEP_NOTIFY_PPM_QUERY_PERF_CAPABILITIES notification. This notification is sent at PASSIVE_LEVEL. It may be sent while executing on any processor.

PEP_NOTIFY_PPM_QUERY_DOMAIN_INFO

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_QUERY_DOMAIN_INFO.

Data

A pointer to a PEP_PPM_QUERY_DOMAIN_INFO structure.

Remarks

An optional notification that queries for information about a performance domain. This notification is sent at PASSIVE_LEVEL. It may be sent while executing on any processor.

PEP_NOTIFY_PPM_RESUME_FROM_SYSTEM_STATE

Handle

A PEPHANDLE structure containing the device handle of the PEP for the target processor. If the notification does not target a specific processor, this will be NULL.

Notification

The value PEP_NOTIFY_PPM_RESUME_FROM_SYSTEM_STATE.

Data

A pointer to a PEP_PPM_RESUME_FROM_SYSTEM_STATE structure.

Remarks

An optional notification that notifies the PEP that the system has just resumed from a system power state. This notification is sent to all processors simultaneously just before processors are released to resume passive level work. This notification is sent at DISPATCH_LEVEL, with all processors at dispatch. This notification is always executed on the target processor.

PPM power control codes

12/5/2018 • 2 minutes to read • Edit Online

The power control codes described in this topic are used by platform extension plug-ins (PEPs). A power control request is similar to an I/O control request (IOCTL). Unlike an IOCTL, however, a power control request is sent directly to the Window power management framework (PoFx) and is not observed by other device drivers in the device stack.

The following are the PPM power control codes:

CODE	SYNTAX	DESCRIPTION
PEP_PPM_POWER_CONTROL_QUERY_PARKING_PAGE	// {38BD8901-AB20-4908-ABAA-AC34674BDFF3} DEFINE_GUID(PEP_PPM_POWER_CONTROL_QUERY_PARKING_PAGE, 0x38bd8901, 0xab20, 0x4908, 0xab, 0xaa, 0xac, 0x34, 0x67, 0x4b, 0xdf, 0xf3);	Code is used by the PEP to query the Windows power management framework (PoFx) for information about the parking page assigned to a processor. To determine the parking page for a processor, the platform extension plug-in (PEP) for this processor submits a PEP_PPM_POWER_CONTROL_QUERY_PARKING_PAGE power control request to PoFx. To initiate this power control request, the PEP first calls the RequestWorker routine to inform PoFx that the PEP has a work item to submit. PoFx responds to this call by sending a PEP_DPM_WORK notification to the PEP. The PEP responds by submitting a power control work request for the parking page information. This request includes a PEP-allocated PEP_WORK_INFORMATION structure in which the WorkType member is set to PepWorkRequestPowerControl, and the PowerControl member points to a PEP-allocated PEP_WORK_POWER_CONTROL structure. The PowerControlCode member of the PEP_WORK_POWER_CONTROL structure is set to PEP_PPM_POWER_CONTROL_QUERY_PARKING_PAGE. The InBuffer member of this structure must be NULL, and the OutBuffer member must point to a PEP-allocated PEP_PPM_CONTEXT_QUERY_PARKING_PAGE structure. In response to this power control request, PoFx writes the virtual and physical addresses of the parking page to the

CODE	SYNTAX	DESCRIPTION
		PEP_PPM_CONTEXT_QUERY_PARKING_PAGE structure.
		The PEP_PPM_POWER_CONTROL_QUERY_PARKING_PAGE power control request is ARM-specific and is not supported for x86 and x64 processors. In an ARM multiprocessor system, a parking page is a 4-kilobyte block of memory that the operating system uses as a mailbox to control a processor that is starting up from an idle state. A PEP might use some part of the mailbox to store processor-specific context data. For more information, see the document titled "Multiprocessor Startup for ARM Platforms" at https://www.acpica.org/related-documents.

CODE	SYNTAX	DESCRIPTION
GUID_PPM_PERF_CONSTRAINT_CHANGE	// {29181FA1-4BF3-4c2e-B314-A6D226322B00} DEFINE_GUID(GUID_PPM_PERF_CONSTRAINT_CHANGE, 0x29181fa1, 0x4bf3, 0x4c2e, 0xb3, 0x14, 0xa6, 0xd2, 0x26, 0x32, 0x2b, 0x0);	Code is used by the PEP to notify the Windows power management framework (PoFx) that the processor's performance limits must change to accommodate external constraints (power budgeting, thermal constraints, power source, and so on). No input or output buffer is used with this control code. To initiate this power control request, the PEP first calls the RequestWorker routine to inform PoFx that the PEP has a work item to submit. PoFx responds to this call by sending a PEP_DPM_WORK notification to the PEP. The PEP responds by submitting a power control work request for a performance constraint change. This request includes a PEP-allocated PEP_WORK_INFORMATION structure in which the WorkType member is set to PepWorkRequestPowerControl, and the PowerControl member points to a PEP-allocated PEP_WORK_POWER_CONTROL structure. The PowerControlCode member of the PEP_WORK_POWER_CONTROL structure is set to GUID_PPM_PERF_CONSTRAINT_CHANGE. Both the InBuffer and OutBuffer members of this structure must be NULL. In response to this power control request, PoFx will send a PEP_NOTIFY_PPM_PERF_CONSTRAINTS notification to the PEP to get the new processor performance limits.

Power Management Responsibilities for Drivers

6/25/2019 • 2 minutes to read • Edit Online

Drivers that support power management are responsible for:

Reporting device power capabilities during PnP enumeration.

Setting device object flags for power management.

Handling power IRPs sent by the power manager or a driver.

Powering up a device as soon as needed after system start-up or idle shutdown.

Powering down a device at system shutdown time or putting it to sleep when idle.

Enabling device wake-up, if the device supports wake-up capabilities.

Managing device performance states, if the device supports decreasing performance or features to reduce power consumption.

Not every driver in every device stack performs all of these tasks. Typically, the bus driver reports capabilities, sets flags, and manipulates the physical device, and the device power policy manager (usually the function driver) issues requests to put the device to sleep and to enable wake-up.

With few exceptions, drivers power on and power off their devices, and they enable devices for wake-up in response to power IRPs, that is, IRPs with the major code **IRP_MJ_POWER**. Power IRPs can be sent by the power manager and, in some cases, by a driver.

Reporting Device Power Capabilities

6/25/2019 • 2 minutes to read • Edit Online

During enumeration, drivers report device-specific information in response to a PnP **IRP_MN_QUERY_CAPABILITIES** request. Along with other such information, drivers report a device's power management capabilities in the **DEVICE_CAPABILITIES** structure. Typically, the bus driver fills in this structure.

Higher-level drivers should set an *IoCompletion* routine for the query-capabilities IRP so that they can make a local copy of the structure and ensure that it contains appropriate values. As a general rule, higher-level drivers should not change these values. However, if a change is necessary, a driver can further restrict device capabilities but cannot add to them. In other words, a driver can make the rules more restrictive but cannot loosen them.

After the IRP is complete and all drivers' completion routines have been run, the structure is cached and a driver cannot change its contents.

The following members of the **DEVICE_CAPABILITIES** structure pertain to power management:

DeviceD1 and DeviceD2

WakeFromD0, WakeFromD1, WakeFromD2, and WakeFromD3

DeviceState

SystemWake

DeviceWake

D1Latency, D2Latency, and D3Latency

DeviceD1 and DeviceD2

6/25/2019 • 2 minutes to read • Edit Online

The **DeviceD1** and **DeviceD2** members of **DEVICE_CAPABILITIES** indicate whether the device hardware supports these device power states. Each is a single bit, which is set if the device supports the state and is clear if the device does not support the state. The operating system assumes that all devices support the D0 and D3 device power states.

WakeFromD0, WakeFromD1, WakeFromD2, and WakeFromD3

6/25/2019 • 2 minutes to read • Edit Online

Each of these **DEVICE_CAPABILITIES** structure members indicates whether the device can awaken in response to an external signal that arrives when the device is in the specified state.

For a device that supports all four device power states (D0, D1, D2, D3) but can awaken only from states D0 and D1, the **WakeFromD0** and **WakeFromD1** bits are set, and the **WakeFromD2** and **WakeFromD3** bits are clear.

DeviceState

6/25/2019 • 2 minutes to read • Edit Online

The **DeviceState** member of **DEVICE_CAPABILITIES** is an array of **DEVICE_POWER_STATE** values that are indexed by **SYSTEM_POWER_STATE** values ranging from **PowerSystemWorking** to **PowerSystemShutdown**. Each element of the array contains the maximum (highest-powered) device power state that the device can support for the system power state denoted by the index, or **PowerDeviceUnspecified** if the system power state is not supported.

For example, on a system that supports only S0, S4, and S5 system power states, the **DeviceState** array for a device that supports only the D0 and D3 states contains the values shown in the following table.

DEVICESTATE ELEMENT	VALUE
DeviceState[**PowerSystemWorking**]	**PowerDeviceD0**
DeviceState[**PowerSystemSleeping1**]	**PowerDeviceUnspecified**
DeviceState[**PowerSystemSleeping2**]	**PowerDeviceUnspecified**
DeviceState[**PowerSystemSleeping3**]	**PowerDeviceUnspecified**
DeviceState[**PowerSystemHibernate**]	**PowerDeviceD3**
DeviceState[**PowerSystemShutdown**]	**PowerDeviceD3**

On a system that supports all of the system power states, the following table lists the values that the array would contain for a device that must be in the D2 state or lower whenever the system goes to any intermediate sleep state and in the D3 state when the system hibernates.

DEVICESTATE ELEMENT	VALUE
DeviceState[**PowerSystemWorking**]	**PowerDeviceD0**
DeviceState[**PowerSystemSleeping1**]	**PowerDeviceD2**
DeviceState[**PowerSystemSleeping2**]	**PowerDeviceD2**
DeviceState[**PowerSystemSleeping3**]	**PowerDeviceD2**
DeviceState[**PowerSystemHibernate**]	**PowerDeviceD3**

DEVICESTATE ELEMENT	VALUE
DeviceState[**PowerSystemShutdown**]	**PowerDeviceD3**

Note that the entries in the **DeviceState** array signify the highest device power state that the device can support for the corresponding system power state. In the preceding example, the device could be in state D3 for any system power state, state D2 for system power states **PowerSystemWorking** through **PowerSystemSleeping3**, and state D1 for system state **PowerSystemWorking**.

The bus driver or ACPI filter sets these values based on the capabilities of the parent device node.

As a general rule, higher-level drivers should not change these values. However, in the rare circumstances in which such a change is necessary, a driver can specify a lower (less-powered) state than the bus driver or ACPI filter originally returned. For example, assume that **DeviceState**[**PowerSystemSleeping1**] maps to **PowerDeviceD2**, as in the table above. A driver can change this value to **PowerDeviceD3**, but not to **PowerDeviceD1** or **PowerDeviceD0**.

SystemWake

6/25/2019 • 2 minutes to read • Edit Online

The **SystemWake** member of **DEVICE_CAPABILITIES** contains the lowest (least-powered) system power state from which the device can wake the system, or **PowerSystemUnspecified** if the device cannot wake the system.

The bus driver sets this value at when it enumerates the device. A higher-level driver can change the value to a higher-powered state but cannot change it to a lower-powered state. For example, if the bus driver sets **SystemWake** to S3 but a driver further up the device stack supports wake-up only from S2, the higher-level driver can change the value to S2. If a driver changes **SystemWake**, it might also have to change **DeviceWake**, as explained in the next section.

Drivers rarely need to propagate changed values back down the device stack. Because changes make the device capabilities more restrictive, lower drivers do not see requests that they cannot handle. In the previous example, a higher-level driver fails any request to wake the system from a lower-powered state than S2, so lower drivers never see such a request. However, if a lower driver must be aware of any changes, it can send a PnP **IRP_MN_QUERY_CAPABILITIES** to its own device stack during its processing of an **IRP_MN_START_DEVICE**.

If both the **SystemWake** and **DeviceWake** members are nonzero (that is, not **PowerSystemUnspecified**), then the device and its drivers support wake-up on this system.

On non-ACPI hardware, this member always contains zero (**PowerSystemUnspecified**).

DeviceWake

6/25/2019 • 2 minutes to read • Edit Online

The **DeviceWake** member of **DEVICE_CAPABILITIES** contains the lowest (least-powered) device power state from which the device can signal a wake event, or **PowerDeviceUnspecified** if the device cannot wake in response to an external signal.

The bus driver sets this value. A higher-level driver can change the value to a higher-powered state. For example, if the bus driver sets **DeviceWake** to D3 but a driver further up the device stack supports wake-up only from D2, the higher-level driver can change the value to D2.

Note that if a driver changes **DeviceWake**, it might also have to change **SystemWake** to avoid conflicts with the system-to-device mappings in the **DeviceState** array. For example, assume that the bus driver sets the following:

- **DeviceState**[**PowerSystemSleeping1**] = **PowerDeviceD1**
- **DeviceState**[**PowerSystemSleeping2**] = **PowerDeviceD3**
- **DeviceWake** = **PowerDeviceD3**
- **SystemWake** = **PowerSystemSleeping2**

If a higher-level driver determines that its device cannot wake the system from D3, but only from D2 or higher, it can change **DeviceWake** to D2. However, this change causes the mapping from S2 to D3 to be impossible. Remember that the **DeviceState** array lists the highest device power state a device can support for a given system power state. If the system power state in the example is **PowerSystemSleeping2**, the device power state cannot be **PowerDeviceD2**. To eliminate this problem, the driver must also change **SystemWake** to **PowerSystemSleeping1**. The same is true for the **WakeFromD***x* and **DeviceD***x* settings. A driver must ensure that any changes it makes to **SystemWake** or **DeviceWake** do not conflict with the **WakeFromD***x* and **DeviceD***x* values. The values of *WakeFromDx* and **DeviceD***x* reflect hardware characteristics that a driver cannot change.

If both the **SystemWake** and **DeviceWake** members are nonzero (that is, not **PowerSystemUnspecified**), then the device and its drivers support wake-up on this system.

On non-ACPI hardware, the **DeviceWake** member contains zero (**PowerSystemUnspecified**).

D1Latency, D2Latency, and D3Latency

6/25/2019 • 2 minutes to read • Edit Online

The **D1Latency**, **D2Latency**, and **D3Latency** members of **DEVICE_CAPABILITIES** contain the approximate time, in 100-microsecond units, that the device requires to return to the D0 state from each of the sleeping states. A driver should specify a latency time of zero for any device power state that it does not support.

Setting Device Object Flags for Power Management

6/25/2019 • 2 minutes to read • Edit Online

In its *AddDevice* routine, each driver creates a device object (filter device object (DO), functional device object (FDO), or physical device object (PDO)) and sets the DO_*XXX* flags in the device object to describe the device attributes and driver configuration. The following device object flags pertain to power management.

FLAG	DESCRIPTION
DO_POWER_INRUSH	Indicates that the current drawn by the device surges when the device is first turned on. This surge or "inrush" lasts for a short period, after which the current drawn by the device falls to a lower operating level.
DO_POWER_PAGABLE	Indicates that the driver is pageable. Starting with Windows 2000, drivers that can be paged must set the DO_POWER_PAGABLE flag. The power manager calls such drivers at IRQL = PASSIVE_LEVEL. For more information about pageable drivers, see Making Drivers Pageable.

The device object flags are typically set by the bus driver when it creates the PDO for the device. However, some function drivers might need to alter the values of these flags as part of their *AddDevice* routines. Starting with Windows Vista, the operating system does not require that all device objects within a device stack have the same power-related flags set. However, in Windows Server 2003, Windows XP, and Windows 2000, all the device objects in a device stack should have the same power-related flags set.

Starting with Windows 2000, drivers of devices that are in the paging path must not set the DO_POWER_PAGABLE flag. A driver is in the "paging path" if it participates in I/O operations on the paging file. Drivers that do not set this flag must be callable at IRQL = DISPATCH_LEVEL. For more information, see Constraints on Dispatch Routines.

In general, drivers should not alter the bus driver's value for the DO_POWER_PAGABLE flag, and a driver must never set this flag if a lower-level driver has cleared it. When handling transitions involving PnP paging requests (typically in response to an **IRP_MJ_PNP** with **IRP_MN_DEVICE_USAGE_NOTIFICATION** request), a storage driver must carefully sequence its setting and clearing of the flag.

Drivers for devices that require an inrush of power at start-up must set the DO_POWER_INRUSH flag in the device object before clearing the DO_DEVICE_INITIALIZING flag. Only one driver in the device stack, typically the bus driver (PDO), needs to set the DO_POWER_INRUSH flag for the device. The flag notifies the power manager that such devices must be powered up one at a time, in sequence with other such devices, to avoid overloading the power supply. The power manager ensures that only one power inrush IRP is active anywhere in the system at any given time.

Starting with Windows Vista, drivers can set both the DO_POWER_PAGABLE flag and the DO_POWER_INRUSH flag. In Windows Server 2003, Windows XP, and Windows 2000, drivers cannot set both the DO_POWER_PAGABLE flag and the DO_POWER_INRUSH flag.

Handling Power IRPs

6/25/2019 • 2 minutes to read • Edit Online

Drivers handle power IRPs in a *DispatchPower* routine. All power management requests have the major IRP code **IRP_MJ_POWER** and one of the following minor codes:

IRP_MN_QUERY_POWER — Queries to determine whether changing power state is feasible

IRP_MN_SET_POWER — Requests a change from one power state to another

IRP_MN_WAIT_WAKE — Requests that a device be enabled to wake itself or the system

IRP_MN_POWER_SEQUENCE — Requests information to optimize power restoration to a particular device

Support for **IRP_MN_SET_POWER** and **IRP_MN_QUERY_POWER** is required. All drivers must be prepared to handle these IRPs.

Support for **IRP_MN_WAIT_WAKE** is required for all drivers in the device stack for any device that can awaken in response to an external signal. A driver sends this IRP to enable the device for wake-up.

Support for **IRP_MN_POWER_SEQUENCE** is optional. This IRP provides an optimization for devices that take a long time to restore power.

A power IRP can specify a system power operation or a device power operation. Power IRPs for the system and power IRPs for individual devices take slightly different paths through a device stack, as explained in the following sections.

Power IRPs for the System

6/25/2019 • 2 minutes to read • Edit Online

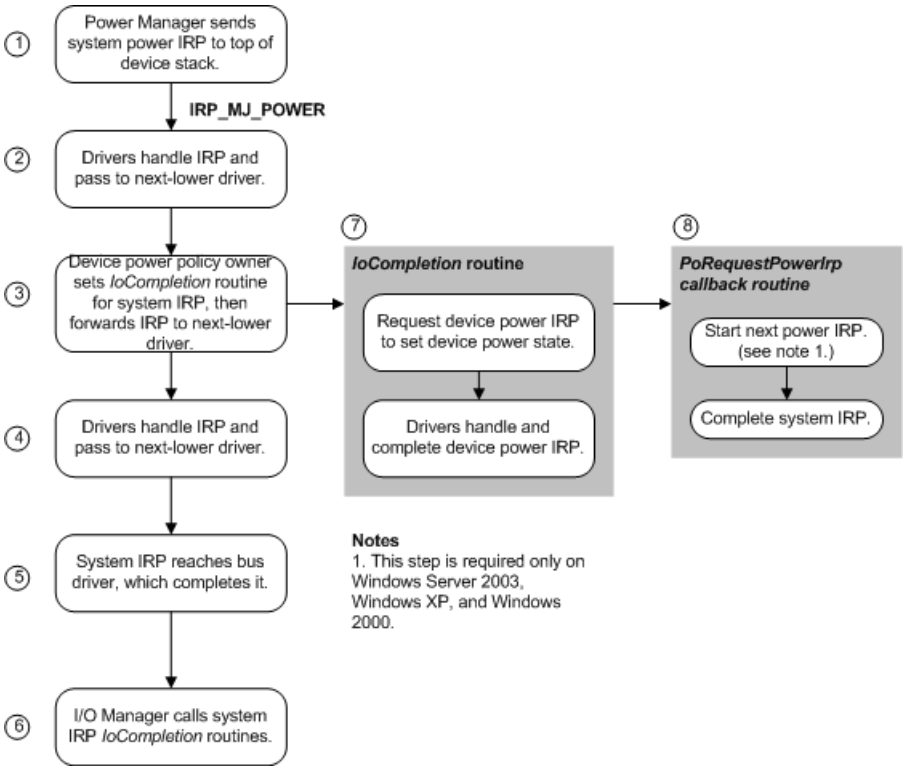
A *system power IRP* specifies major IRP code **IRP_MJ_POWER**, one of the minor power IRP codes listed below, and the value **SystemPowerState** in the **Power.Type** member of the IRP stack. Only the power manager can send such an IRP; a driver cannot send a system power IRP.

The power manager sends a system power IRP for one of the following reasons:

- To change the system power state in response to an idle time-out, a change in system activity, a user request, or an expiring battery (**IRP_MN_SET_POWER**)
- To query devices to determine whether the system can go to sleep (**IRP_MN_QUERY_POWER**)
- To reaffirm the current system power state after a query (**IRP_MN_SET_POWER**)

The power manager sends **IRP_MN_QUERY_POWER** and **IRP_MN_SET_POWER** requests on behalf of the system. A driver can fail an **IRP_MN_QUERY_POWER** request but cannot fail **IRP_MN_SET_POWER**.

For example, to change the system power state, the power manager sends a system power IRP to the top driver in the stack at each device node of the device tree. The following figure shows how drivers within a single device stack handle a system power IRP.



As the previous figure shows:

1. The power manager calls the I/O manager to send a system power IRP to each leaf node in the device tree.
2. Drivers handle the IRP if possible, set *IoCompletion* routines if necessary, and call **IoCallDriver** (Windows 7 and Windows Vista) or **PoCallDriver** (Windows Server 2003, Windows XP, and Windows 2000) to forward the IRP down the stack. If a driver must fail the IRP, the driver does so immediately and completes the IRP. Drivers can fail **IRP_MN_QUERY_POWER** IRPs, but must not fail **IRP_MN_SET_POWER** IRPs that set the system power state.

3. When the driver that owns power policy for the device receives the IRP, that driver sets an *IoCompletion* routine for the system IRP and then forwards the IRP.

4. Any other drivers in the stack handle the IRP if possible, set *IoCompletion* routines if necessary, and forward the IRP to the next-lower driver, as in step 2.

5. Eventually, the bus driver receives and completes the system IRP.

6. The I/O manager calls any *IoCompletion* routines that were set as drivers passed the system IRP down the device stack.

7. In its *IoCompletion* routine, the device power policy owner calls **PoRequestPowerIrp** to send a device power IRP, specifying a device power state that is valid for the system power state in the system IRP. The driver sets a callback routine to be invoked when the device power IRP completes.

   If necessary, the driver consults the **DeviceState** member in its cached copy of the **DEVICE_CAPABILITIES** structure (see Reporting Device Power Capabilities) to determine which device power states correspond to the system power state in the IRP.

8. After the device IRP is complete and any device IRP completion routines have run, the power policy owner's callback routine is invoked. In the callback routine, the driver copies its returned status into the system IRP. In Windows Server 2003, Windows XP, and Windows 2000, the callback calls **PoStartNextPowerIrp** to start the next power IRP. However, in Windows 7 and Windows Vista, calling **PoStartNextPowerIrp** is not required and such a call performs no power management operation. Finally, the callback calls **IoCompleteRequest** to complete the system IRP.

For further information, see Handling System Power State Requests.

Because some devices require an inrush of current when they power on, system inrush power IRPs are handled synchronously and serially throughout the system. Only one such IRP can be active at a time. For further information, see Calling IoCallDriver vs. Calling PoCallDriver.

Power IRPs for Individual Devices

6/25/2019 • 2 minutes to read • Edit Online

A *device power IRP* specifies major IRP code **IRP_MJ_POWER**, one of the minor power IRP codes listed below, and the value **DevicePowerState** in the **Power.Type** member.

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

IRP_MN_WAIT_WAKE

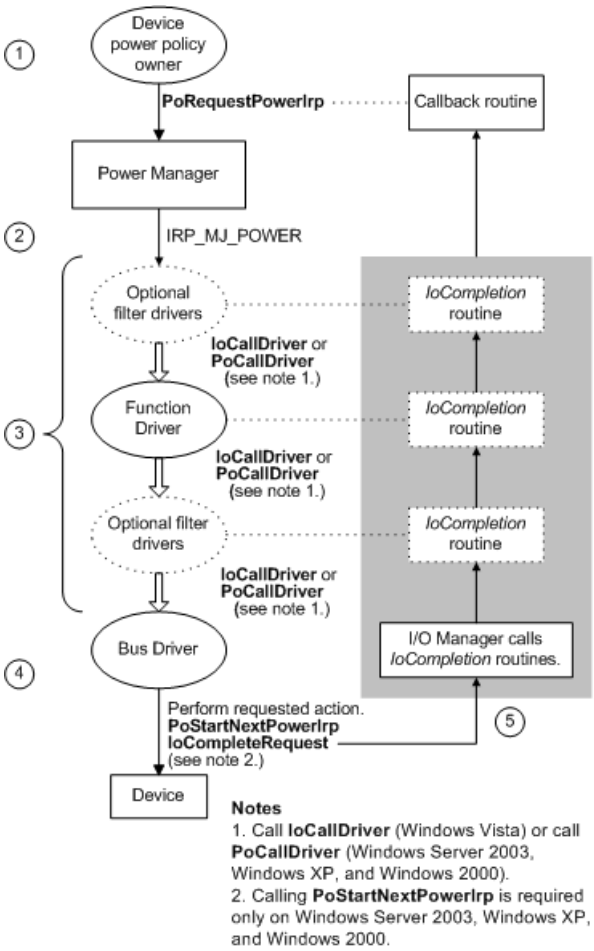
IRP_MN_POWER_SEQUENCE

All drivers in a device stack receive such IRPs; normally, only the device power policy manager can send these IRPs. However, the power manager can send a device power IRP when performing idle detection on behalf of a device, as explained in Using Power Manager Routines for Idle Detection.

A driver sends a device power IRP for any of the following reasons:

- To query or change the device power state in response to a system power IRP
- To put the device in a sleep state to conserve power
- To return the device to the working state after it has been asleep
- To enable the device to awaken in response to an external signal
- To get a power sequence value when powering up a device

The following figure shows the sequence of steps that occur to send, forward, and complete a device power IRP.



As the previous figure shows, a device power IRP is sent, forwarded, and completed in the following steps:

1. The device power policy owner calls **PoRequestPowerIrp** to allocate a device power IRP, specifying the PDO that is the target of the IRP and a callback routine to be invoked when the IRP is complete.

2. The power manager allocates a device power IRP and sends it to the top driver in the device stack for the target PDO.

3. The driver performs the following actions:

   - Sets an *IoCompletion* routine if one is necessary.
   - Calls **PoStartNextPowerIrp** (Windows Server 2003, Windows XP, and Windows 2000) if a completion routine is not used. Beginning with Windows Vista, this call is not required and such a call performs no power management operation.
   - Calls **IoCallDriver** (Windows 7 and Windows Vista) or calls **PoCallDriver** (Windows Server 2003, Windows XP, and Windows 2000) to pass the IRP down to the next-lower driver.

   Each driver in the stack does this until the IRP reaches the bus driver. If a driver must fail the IRP, it should do so immediately and call **IoCompleteRequest**.

4. The bus driver, which maintains the device PDO, performs the requested action, and then calls **IoCompleteRequest** to complete the IRP. A bus driver can fail a device power-up IRP if a device is removed or in the process of being removed.

5. The I/O manager calls *IoCompletion* routines that were set by drivers as they passed the IRP down the stack. After all the *IoCompletion* routines have been called, the callback routine is run.

For more information about device power IRPs, see Managing Power for Individual Devices and Supporting Devices that Have Wake-Up Capabilities. For details on the power sequence IRP, see **IRP_MN_POWER_SEQUENCE**.

Power Management Minor IRPs

6/25/2019 • 2 minutes to read • Edit Online

All power IRPs have the major code **IRP_MJ_POWER** and one of the following minor codes, indicating a specific power management request:

IRP_MN_POWER_SEQUENCE

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

IRP_MN_WAIT_WAKE

This section provides reference information for the individual IRPs in alphabetical order. For more information about when the IRPs are sent and how drivers should handle them, see Power Management. The Power Management section also includes a general introduction to power management and terminology.

IRP_MN_POWER_SEQUENCE

6/25/2019 • 2 minutes to read • Edit Online

This IRP returns the power sequence values for a device.

Major Code

IRP_MJ_POWER When Sent

A driver sends this IRP as an optimization to determine whether its device actually entered a specific power state. Support for this IRP is optional.

To send this IRP, a driver must call **IoAllocateIrp** to allocate the IRP, specifying the major IRP code **IRP_MJ_POWER** and minor IRP code **IRP_MN_POWER_SEQUENCE**. The driver must then call **IoCallDriver** (Windows Vista) or **PoCallDriver** (Windows Server 2003, Windows XP, and Windows 2000) to pass the IRP to the next lower driver. The power manager cannot send this IRP.

Senders of this IRP must be running at IRQL <= DISPATCH_LEVEL.

Input Parameters

None.

Output Parameters

Parameters.PowerSequence points to a **POWER_SEQUENCE** structure with the following members:

SequenceD1
Number of times the device has been in power state D1 or lower.

SequenceD2
Number of times the device has been in power state D2 or lower.

SequenceD3
Number of times the device has been in power state D3.

The sequence values track the minimum number of times a device has been in the corresponding power state or a lower power state.

The bus driver increments the values in **SequenceD1**, **SequenceD2**, and **SequenceD3** at least each time the device enters in the corresponding power state or a lower power state.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS to indicate that it has returned the requested information, or to STATUS_NOT_IMPLEMENTED to indicate that it does not support this IRP.

Operation

This IRP returns the power sequence values for a device. Bus drivers can optionally handle it; function and filter drivers can optionally send it.

For a device that takes a long time to change state, this IRP provides a useful optimization. Every time the device

changes its power state, its bus driver increments the sequence value for that power state. The bus driver initializes the sequence values at boot time and continually increments them thereafter; they need not be reset to zero.

A device policy owner can send this IRP once to get the sequence values before shutting off the device and once again to get new values when restoring power to the device. By comparing the two sets of values, the driver can determine whether the device actually entered the lower-powered state. If the device did not lose power, the driver can avoid a time-consuming reinitialization when the device returns to the D0 state.

For example, if the device takes a long time to restore power upon reaching the D2 state, the driver can store the **SequenceD2** value before it sets the device state to D2 or lower. Later, when power is being restored to the device, the driver can compare the new **SequenceD2** value with its stored value to determine whether the device state actually dropped below D2. If the values match, the device did not actually enter power state D2 or a lower state, and the driver can avoid reinitializing the device.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

IRP_MN_QUERY_POWER

6/25/2019 • 4 minutes to read • Edit Online

This IRP queries a device to determine whether the system power state or the device power state can be changed.

Major Code

IRP_MJ_POWER When Sent

The power manager or a device power policy owner sends this IRP to determine whether it can change the system or device power state, typically to go to sleep. A driver must call **PoRequestPowerIrp** to allocate and send this IRP.

The power manager sends this IRP at IRQL = PASSIVE_LEVEL to device stacks that set the DO_POWER_PAGABLE flag in the PDO.

The power manager can send the IRP at IRQL = DISPATCH_LEVEL if the DO_POWER_INRUSH flag is set. Such drivers cannot directly or indirectly access any paged code or data.

Input Parameters

Parameters.Power.Type specifies the type of power state being set, either **SystemPowerState** or **DevicePowerState**.

Parameters.Power.State specifies the power state itself, as follows:

- If **Parameters.Power.Type** is **SystemPowerState**, the value is an enumerator of the **SYSTEM_POWER_STATE** type.
- If **Parameters.Power.Type** is **DevicePowerState**, the value is an enumerator of the **DEVICE_POWER_STATE** type.

Parameters.Power.ShutdownType specifies additional information about the requested transition. Possible values are enumerators of the **POWER_ACTION** type.

Output Parameters

None.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS to indicate that the device can enter the requested state. A driver sets any appropriate failure status to indicate that it cannot enter the requested state.

Operation

The parameters for **IRP_MN_QUERY_POWER** are identical to those for **IRP_MN_SET_POWER**. Rather than notifying drivers of an irrevocable change to the power state, however, **IRP_MN_QUERY_POWER** queries whether the system or a device can enter a particular power state.

A driver must not change the power state of its device in response to an **IRP_MN_QUERY_POWER** request.

After a driver receives an **IRP_MN_QUERY_POWER** request on Windows Server 2003, Windows XP, and

Windows 2000, a driver must call **PoStartNextPowerIrp**, as described in Calling **PoStartNextPowerIrp**. Beginning with Windows Vista, calling **PoStartNextPowerIrp** is not required and such a call performs no power management operation.

IRP_MN_QUERY_POWER for a System Power State

The power manager sends this IRP to ensure that it can change the system power state without disrupting work, such as dropping network connections.

Whenever possible, the power manager queries before sending **IRP_MN_SET_POWER** to request a system sleep state or a normal system shutdown. However, under some critical conditions (such as the user pressing the **Power Off** button or a battery expiring), the power manager might send an **IRP_MN_SET_POWER** request without first sending a query power request. The power manager queries only for sleep states; it never queries before returning to the working state.

When a driver receives a system power query IRP, it should fail the IRP if it cannot support any of the device states that are valid for the queried system state. For more information, see **DeviceState**. Otherwise, the driver should pass the IRP to the next lower driver. The bus driver completes the IRP.

Beginning with Windows Vista, transition to a system sleep state is considered a critical operation. Although a driver might fail a system query-power IRP, the power manager might still change the system power state to a sleep state. After a driver receives a system query-power IRP, the driver should always be prepared for a subsequent change in the system power state.

When a device power policy owner receives a system power query IRP, it should set an *IoCompletion* routine in the IRP before passing it down. In the *IoCompletion* routine, it should send an **IRP_MN_QUERY_POWER** for a device state that is valid for the queried system state. For more information, see Handling a System Query-Power IRP in a Device Power Policy Owner.

When the IRP specifies **PowerSystemShutdown** (S5), the value at **Parameters.Power.ShutdownType** provides a reason for the shutdown. The **ShutdownType** tells the driver whether the system is resetting (**PowerActionShutdownReset**) or powering off indefinitely to reboot later (**PowerActionShutdownOff**). For drivers of most devices, the difference is inconsequential. However, for certain devices, such as a video streaming device that performs DMA, a driver might opt to power down its device when the system is resetting, thus stopping any ongoing I/O.

On Microsoft Windows 2000 and later systems, the value at **ShutdownType** can also be **PowerActionShutdown**. In this case, the driver cannot tell what type of shutdown is requested and should therefore proceed as for a reset.

If a driver fails an **IRP_MN_QUERY_POWER** request for a system power state, the power manager typically responds by issuing an **IRP_MN_SET_POWER** IRP. Usually, this IRP will reaffirm the current system state. However, it is possible that drivers might receive an **IRP_MN_SET_POWER** to the queried state or to some other intermediate state. Drivers should be prepared to handle these situations.

IRP_MN_QUERY_POWER for a Device Power State

A device power policy owner sends this IRP to its stack in response to a system **IRP_MN_QUERY_POWER** request.

If a driver can put its device in the requested device state, it sets **IoStatus.Status** to STATUS_SUCCESS and passes the IRP down to the next lower driver, and so forth until the IRP reaches the bus driver. If any driver in the stack must fail the IRP, that driver should complete the IRP immediately by calling **IoCompleteRequest** and returning a failure status. Drivers that fail the IRP do not pass it further down the stack.

By returning STATUS_SUCCESS, the driver guarantees that it will not start any operation that would change its ability to set the requested power state. The driver should queue any IRPs that require such operations until it completes a set-power IRP that returns the device to an acceptable power state.

On Windows 2000 and later systems, when the IRP specifies **PowerDeviceD1**, **PowerDeviceD2**, or **PowerDeviceD3**, the value at **Parameters.Power.ShutdownType** provides information about the current system power IRP, if a system power IRP is active. In this case, the value at **ShutdownType** indicates the currently requested system power state, or **PowerActionNone** if a system request is not outstanding. On Windows 98/Me, this field always contains **PowerActionNone** when the IRP requests a device power state.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

PoRequestPowerIrp

PoStartNextPowerIrp

IRP_MN_SET_POWER

6/25/2019 • 8 minutes to read • Edit Online

This IRP notifies a driver of a change to the system power state or sets the device power state for a device.

Major Code

IRP_MJ_POWER When Sent

Either the system power manager or a device power policy owner can send this IRP.

The power manager sends this IRP to notify drivers of a change to the system power state. If a driver has registered its device for idle detection, the power manager sends this IRP to change the power state of an idle device.

A driver that owns power policy sends this IRP to set the device power state for its device. A driver must call **PoRequestPowerIrp** to send this IRP.

The power manager sends this IRP at IRQL = PASSIVE_LEVEL to device stacks that set the DO_POWER_PAGABLE flag in the PDO. Drivers in such stacks can touch paged code or data to complete the request.

The power manager can send the IRP at IRQL = DISPATCH_LEVEL if the DO_POWER_INRUSH flag is set. Such drivers cannot directly or indirectly access any paged code or data.

Input Parameters

The **Parameters.Power.Type** member specifies the type of power state being set, either **SystemPowerState** or **DevicePowerState**.

The **Parameters.Power.State** member specifies the power state itself, as follows:

- If **Parameters.Power.Type** is **SystemPowerState**, the value is an enumerator of the **SYSTEM_POWER_STATE** type.
- If **Parameters.Power.Type** is **DevicePowerState**, the value is an enumerator of the **DEVICE_POWER_STATE** type.

The **Parameters.Power.ShutdownType** member specifies additional information about the requested transition. The possible values for this member are **POWER_ACTION** enumeration values. For more information, see System Power Actions.

Starting with Windows Vista, the **Parameters.Power.SystemPowerStateContext** member is a read-only, partially opaque **SYSTEM_POWER_STATE_CONTEXT** structure that contains information about the previous system power states of a computer. If **Parameters.Power.Type** is **SystemPowerState** and **Parameters.Power.State** is **PowerSystemWorking**, two flag bits in this structure indicate whether a fast startup or a wake-from-hibernation caused the computer to enter the S0 (working) system state. For more information, see Distinguishing Fast Startup from Wake-from-Hibernation.

The following table shows the contents of **IRP_MN_SET_POWER.Parameters.Power.{State|ShutdownType}** and the **CurrentSystemState**, **TargetSystemState**, and **EffectiveSystemState** bit fields in the **SYSTEM_POWER_STATE_CONTEXT** structure for each system power transition. Each row represents one **IRP_MN_SET_POWER**.

TRANSITION	STATE	SHUTDOWN TYPE	CURRENT SYSTEMSTATE	TARGET SYSTEMSTATE	EFFECTIVE SYSTEMSTATE	COMMENTS	
Sleep to...	S3	Sleep	S0	S3	S3		
...Wake	S0	Sleep	S3	S0	S0		
Hybrid Sleep to...	S4	Hibernate	S0	S3	S4	Sleep with hibernation file (Fast S4)	
...Wake	S0	Sleep	S3	S0	S0		
...Wake/Pwr Lost	S0	Sleep	S4	S0	S0		
Hibernate to...	S4	Hibernate	S0	S4	S4		
...Wake	S0	Sleep	S4	S0	S0		
Hybrid Shutdown to...	S4	Hibernate	S0	S5	S4	Apps closed, user logged off as if shutdown (Hiber Boot)	
...Fast Startup	S0	Sleep	S4	S0	S0		
Shutdown to...	S5	Shutdown/ Reset/Off	S0	S5	S5		
...System Boot						No S-IRP for boot up	

Output Parameters

Parameters.Power.SystemContext is reserved for system use.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS to indicate that the device has entered the requested state.

A driver must not fail a request to set the system power state.

Function and filter drivers that are located above a bus driver must not fail a request to set a device power state. The bus driver can fail a device power-up request if the device is removed or in the process of being removed.

Operation

The power manager or a driver can request an **IRP_MN_SET_POWER** IRP. The power manager sends this IRP for one of the following reasons:

- To notify drivers of a change to the system power state
- To change the power state of a device for which the power manager is performing idle detection
- To reaffirm the current system state after a driver fails an **IRP_MN_QUERY_POWER** request for a system power state. For more information, see **IRP_MN_QUERY_POWER**.

A driver that owns device power policy sends **IRP_MN_SET_POWER** to change the power state of its device.

At any given time, the system allows only one such IRP to be active for each device object.

Each driver must pass each power IRP down to the next-lower driver by calling **IoCallDriver** (starting with Windows Vista) or **PoCallDriver** (Windows Server 2003, Windows XP, and Windows 2000). The **PoCallDriver** interface is similar to that of **IoCallDriver**, except that the power management subsystem might delay the IRP before passing it on to the next driver. For example, delays can occur on a **PowerDeviceD0** request if the device requires inrush current and therefore must be powered up serially with another such device.

After a driver receives an **IRP_MN_SET_POWER** request on Windows Server 2003, Windows XP, or Windows 2000, a driver must call **PoStartNextPowerIrp**, as described in Calling PoStartNextPowerIrp. Beginning with Windows Vista, calling **PoStartNextPowerIrp** is not required, and such a call performs no power management operation.

IRP_MN_SET_POWER for System Power States

Only the system power manager can send a system set-power IRP.

A driver must not fail a request to set the system power state.

Whenever possible, the power manager sends **IRP_MN_QUERY_POWER** before sending **IRP_MN_SET_POWER** to request a system sleep state. However, under some conditions (such as the user pressing the **Power Off** button or a battery expiring), the power manager might issue **IRP_MN_SET_POWER** without first querying. The power manager queries only for sleep states; it never queries before powering up.

The **IRP_MN_SET_POWER** request is sent to the top driver in the device stack for a device. The top driver passes the IRP down to the next lower driver and so forth until the IRP reaches the bus driver, which must complete the IRP.

A filter driver typically does not need to act on a system set-power IRP, other than to pass it on.

The device power policy owner, however, sets an*IoCompletion* routine before passing down the IRP. In the *IoCompletion* routine, it sends an **IRP_MN_SET_POWER** request for a device power IRP. For more information, see Handling a System Set-Power IRP in a Device Power Policy Owner.

A system set-power IRP informs drivers that a change to the system power state is imminent and the drivers must prepare for it. However, a driver should not change the power state of its device until it receives an **IRP_MN_SET_POWER** for a *device* power state.

The value at **Parameters.Power.ShutdownType** provides additional information about the pending actions. When the IRP specifies **PowerSystemShutdown** (S5), a driver can determine whether the system is resetting (**PowerActionShutdownReset**) or powering off indefinitely to reboot later (**PowerActionShutdownOff**). For drivers of most devices, the difference is inconsequential. However, for certain devices, such as video streaming devices, a driver might power off the device in order to stop I/O when the system is resetting.

On Windows 2000 and later versions of the operating system, the value at **ShutdownType** can also be **PowerActionShutdown**. In this case, the driver cannot tell what type of shutdown is requested and should therefore proceed as for a reset.

Device Power States

Function and filter drivers that are located above a bus driver must not fail a request to set a device power state.

The bus driver can fail a device power-up request if the device is removed or in the process of being removed.

A driver must set the device into the requested state before completing the IRP.

When the IRP requests a transition to a lower power state, drivers must handle the IRP as it travels down the device stack, saving any context the driver will need to restore the device to the working state. After a bus driver receives an IRP, the driver:

- Saves any context the driver will need to restore the device to the working state.
- Sets the device to the requested power state.
- Calls **PoSetPowerState** to notify the power manager.
- Calls **PoStartNextPowerIrp** to start the next power IRP (Windows Server 2003, Windows XP, and Windows 2000 only).
- Completes the device power IRP.

The driver must complete this IRP in a timely manner. In general, drivers should avoid any delay that a typical user would find noticeably slow. For example, a driver could delay a system state change to flush cached disk or network data, but should not keep a network connection alive or format a tape. For more information, see Passing Power IRPs.

On Windows 2000 and later versions of the operating system, if the IRP specifies **PowerDeviceD1**, **PowerDeviceD2**, or **PowerDeviceD3**, and a system set-power IRP is active, the value at **Parameters.Power.ShutdownType** provides information about the system IRP.

Drivers of devices on the hibernate path should inspect this value. If the IRP requests **PowerDeviceD3** and **ShutdownType** is **PowerActionHibernate**, such a driver should save any context required to restore the device, but should not power down the device; the device will enter the D3 state when the machine loses power.

On Windows 2000 and later versions of the operating system, drivers should not rely on the value at **ShutdownType** if the requested power state is **PowerDeviceD0**.

On Windows 98/Me, if the IRP requests a device power state, the **ShutdownType** is always **PowerActionNone**.

The driver that determines when to power down a device varies depending on the device class.

The driver that determines when to power up a device is almost always a driver that accesses the device registers. The driver must verify that the device is in the D0 state before accessing the device's hardware registers. If the device is not in the D0 state, the driver must call **PoRequestPowerIrp** to send an IRP to power up the device. A driver cannot access its device unless the device is in the D0 state.

When a driver receives a set-power IRP for device state D0, it sets an *IoCompletion* routine and passes the IRP to the next lower driver.

When the IRP reaches the bus driver, that driver applies (or resets) power to the device, calls **PoStartNextPowerIrp** (Windows Server 2003, Windows XP, and Windows 2000 only), and calls **PoSetPowerState** to inform the power manager of the new power state for the device.

After the bus driver completes the power-up IRP, function and filter drivers handle the IRP in their *IoCompletion* routines as it travels back up the device stack. In the *IoCompletion* routine, each driver restores or reinitializes its device context and performs any other required start-up tasks.

For more information, see Handling IRP_MN_SET_POWER for Device Power States.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DEVICE_POWER_STATE

IoCallDriver

IRP_MN_QUERY_POWER

IRP_MN_SET_POWER

PoCallDriver

PoStartNextPowerIrp

PoSetPowerState

PoRequestPowerIrp

SYSTEM_POWER_STATE

SYSTEM_POWER_STATE_CONTEXT

IRP_MN_WAIT_WAKE

6/25/2019 • 4 minutes to read • Edit Online

This IRP enables a driver to awaken a sleeping system or to awaken a sleeping device.

Major Code

IRP_MJ_POWER When Sent

A driver that owns power policy targets this IRP to its PDO to enable its device to awaken in response to an external event, such as an incoming phone call. A driver must call **PoRequestPowerIrp** to send this IRP.

As a general rule, a driver should send this IRP as soon as it determines that its device should be enabled for wake-up. Consequently, drivers for most such devices send this IRP after powering on their devices and before completing the **IRP_MN_START_DEVICE** request.

However, a driver can send the IRP any time the device is in the working state (**PowerDeviceD0**). The device stack must not be in transition; that is, a driver should not send an **IRP_MN_WAIT_WAKE** while any other power IRP is active in its device stack.

A wait/wake IRP does not change the power state of the device or of the system. It simply enables a wake-up signal from the device. When the wake-up signal arrives, the policy owner must call **PoRequestPowerIrp** to send a set-power IRP to return its device to D0.

The driver must be running at IRQL = PASSIVE_LEVEL to send this IRP. However, the IRP can be completed at IRQL = DISPATCH_LEVEL.

Input Parameters

Parameters.WaitWake.PowerState contains the lowest (least-powered) system power state from which the device should be allowed to awaken the system.

Output Parameters

None.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to one of the following:

STATUS_PENDING
The driver received the IRP and is waiting for the device to signal wake-up.

STATUS_INVALID_DEVICE_STATE
The device is in a less-powered state than the **DeviceWake** state specified in the **DEVICE_CAPABILITIES** structure for the device, or the device cannot awaken the system from the **SystemWake** state passed in the IRP.

STATUS_NOT_SUPPORTED
The device does not support wake-up.

STATUS_DEVICE_BUSY
An **IRP_MN_WAIT_WAKE** request is already pending and must be completed or canceled before another

IRP_MN_WAIT_WAKE request can be issued.

STATUS_SUCCESS
The device has signaled a wake event.

STATUS_CANCELLED
The IRP has been canceled.

If a driver must fail this IRP, it completes the IRP immediately and does not pass the IRP to the next-lower driver.

Operation

A driver sends **IRP_MN_WAIT_WAKE** for either of two reasons:

1. To enable its device to awaken a sleeping system in response to an external wake-up signal.

2. To enable its device to awaken from a device sleep state in response to an external wake-up signal.

The IRP must be passed down the device stack to the bus driver for the device, which calls **IoMarkIrpPending** and returns STATUS_PENDING from its *DispatchPower* routine. The IRP remains pending until a wake-up signal occurs or until the driver that sent the IRP cancels it.

Only one wait/wake IRP can be held pending for a PDO at any given time. If a driver already holds a wait/wake IRP for a PDO, it must fail any additional such IRPs with STATUS_DEVICE_BUSY. A driver that enumerates more than one child PDO can have a wait/wake IRP pending for each such PDO.

Each driver sets an *IoCompletion* routine as the IRP travels down the device stack. When the device signals a wake-up event, the bus driver services the wake-up signal and completes the IRP, returning STATUS_SUCCESS. The I/O manager then calls the *IoCompletion* routine of the next higher driver, and so on up the device stack.

When a driver sends a wait/wake IRP, it should specify a callback routine in the **PoRequestPowerIrp** call. In the callback routine, the driver typically services the device. For example, the power policy owner for the device must call **PoRequestPowerIrp** to send an **IRP_MN_SET_POWER** for device state D0.

A driver that acts as the bus driver for one device and the policy owner for a parent device requests an **IRP_MN_WAIT_WAKE** IRP for the parent's device stack when it receives a **IRP_MN_WAIT_WAKE** request from a child PDO. If the driver enumerates more than one child PDO, it should request only one wait/wake IRP for the parent's device stack no matter how many child PDOs send wait/wake requests. Instead, such a driver should keep an internal count of wait/wake IRPs, incrementing the count each time it receives a request and decrementing the count each time it completes a request. If the count is nonzero after it has completed a wait/wake IRP, the driver should send another wait/wake IRP to its device stack to "rearm" itself for wake-up. For more information, see Understanding the Path of Wait/Wake IRPs through a Device Tree.

To cancel an **IRP_MN_WAIT_WAKE**, a driver calls **IoCancelIrp**. Only the driver that originated the IRP can cancel it. A driver cancels a pending **IRP_MN_WAIT_WAKE** when any of the following occurs:

- The driver receives a PnP IRP that stops or removes the device.
- The system is going to sleep and the device wake signal must not awaken it.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

PoRequestPowerIrp

Powering Up a Device

6/25/2019 • 2 minutes to read • Edit Online

When a bus driver handles a PnP **IRP_MN_START_DEVICE** request for one of its child devices, it should power on the device and call **PoSetPowerState** to report the device power state to the power manager. Powering on the device is an implicit part of device start-up. The device power policy owner does not send an **IRP_MN_SET_POWER** request for **PowerDeviceD0**, so drivers should not expect to receive these IRPs at start-up.

When a device has been powered down to conserve power, its drivers should power it up when an I/O request arrives. In this case, the device power policy owner must send an **IRP_MN_SET_POWER** to return the device to the working state. When the IRP completes, the drivers for the device stop queuing I/O and begin to process requests off the queue.

Powering Down a Device

6/25/2019 • 2 minutes to read • Edit Online

Unless a device is enabled for wake-up, its drivers power it off when the system shuts down. Devices must always be powered off upon removal or surprise removal.

When a device is removed, the Plug and Play manager sends an **IRP_MN_REMOVE_DEVICE** request to the device stack. In response to this IRP, the drivers for the device should ensure that the device powers down. Powering down the device is an implicit part of removal handling; the device power policy owner is not required to send an **IRP_MN_SET_POWER** for **PowerDeviceD3**.

As drivers handle the **IRP_MN_REMOVE_DEVICE** request, they wait for pending I/O to complete, perform any necessary removal processing, call **PoSetPowerState** to notify the power manager that the device is in state D3, and delete the device objects they created for this device. Typically, the bus driver turns off power to the device.

If a device is unexpectedly removed from a Windows 2000 or later operating system, the Plug and Play manager sends an **IRP_MN_SURPRISE_REMOVAL** request to the top of the corresponding device stack. In response to this IRP, the drivers for the device should perform surprise removal processing, as described in Handling an IRP_MN_SURPRISE_REMOVAL Request.

At system shutdown, the power manager sends an **IRP_MN_SET_POWER** for a system power state (either S4 or S5). When the device power policy owner receives this IRP, it should send an **IRP_MN_SET_POWER** for **PowerDeviceD3** so that lower drivers can finish their work and power down the device.

A driver can optionally perform idle detection for its device, or can request that the power manager perform idle detection, so that the device can be powered down when not in use. For further information, see Detecting an Idle Device.

Enabling Device Wake-Up

6/25/2019 • 2 minutes to read • Edit Online

If a device supports wake-up, its power policy owner must be able to enable and disable wake-up for the device. A driver enables wake up by sending an **IRP_MJ_POWER** request with minor function code **IRP_MN_WAIT_WAKE** and disables wake-up by canceling a previously sent **IRP_MN_WAIT_WAKE**. A device can have only one **IRP_MN_WAIT_WAKE** request pending at a time.

To determine whether its device supports wake-up, the device power states from which it can signal wake-up, and the system power states from which the device can wake the system, a driver checks the **SystemWake**, **DeviceWake**, and **WakeFromD***x* members in the **DEVICE_CAPABILITIES** structure.

For more information about enabling, disabling, and responding to wake-up signals in a driver, see Supporting Devices that Have Wake-Up Capabilities.

Managing Device Performance States

6/25/2019 • 5 minutes to read • Edit Online

Windows Vista features an enhanced power management infrastructure that makes it possible for driver stacks to better manage the power policy of their devices. Drivers can register to be notified when system-defined power settings change or when system power events occur. A device power policy owner can use these notifications to appropriately adjust the power usage of its devices. In addition, you can create custom power settings that provide access to device-specific power and performance features, which can be tightly-integrated into system power policy. The following are the two primary approaches to integrate device performance states and power-saving behaviors with system power policy.

Creating Custom Power Settings for a Device

You can define custom power settings that can be used to configure device performance states or power-saving behaviors. Information about the custom power settings is saved and managed by the power manager. Other components in the system—such as device drivers, services, or applications—can register to be notified when the value of a custom power setting changes. In general, devices that have the capability to tradeoff performance with power consumption should have corresponding custom power settings. Creating custom power settings is the most flexible approach to tightly integrate power consumption with system power policy and provides the following additional benefits:

- A custom user interface is not required to make custom power settings accessible to a user. All power settings, including custom power settings, are presented to the user on the **Advanced Settings** page of the **Power Options** Control Panel.
- Users and system administrators can easily script the configuration of custom power settings by using Powercfg.exe, the power management command-line tool.
- Optionally, a system administrator can create an administrative template (.ADM) file that enables group policy-based configuration of new power settings.

An individual power setting contains all of the information that is required to uniquely identify, name, describe, and provide values for the power setting. Each power setting is defined with a GUID, a setting name, a description, and default settings for AC and DC power schemes. A custom power setting can be created statically for a device, by using an **INF AddPowerSetting directive**, or dynamically, by calling the Win32 power management functions that are included in the power management reference that is provided with Microsoft Windows SDK documentation.

Drivers call **PoRegisterPowerSettingCallback** to register a callback routine that the power manager calls to notify the driver of a change to a power setting. When the setting changes, the power manager calls the callback routine and passes the new setting value. Drivers can then take the action that is appropriate for the power setting. Each setting is identified by the GUID of the power setting. The system-defined GUIDs for power settings are defined in Wdm.h and Ntpoapi.h.

For example, to be notified when monitor power is turned on or off, a driver calls **PoRegisterPowerSettingCallback**, supplying the GUID that identifies the monitor power setting (GUID_MONITOR_POWER_ON) and a pointer to the callback routine that the power manager calls when the value of the monitor power setting changes.

Registering to be Notified of a Change to the Active Power Scheme, Power Scheme Personality, or Power

Source

The personality of the active power scheme conveys the user's intent for the overall power saving behavior of the system. Every power scheme, including custom schemes, have a personality that indicates the overall intention of the scheme. This enables additional custom power schemes to be created while still providing all of the benefits of knowing the intent of the scheme. Windows Vista includes the following three system-defined power schemes and their corresponding personalities.

Maximum power savings
Reduces performance to minimize power consumption.

Automatic (balanced)
Lets the system choose the best power state level based on overall power consumption.

Maximum performance
Provides maximum performance regardless of power consumption.

The power source can be either an AC or a DC power source.

A device power policy owner can use information about the active power scheme, power scheme personality, and power source to adjust device power policy. For example, a device power policy owner might aggressively power down a device if the power scheme personality changes to **Maximum Power Savings**. However, if the power scheme personality changes to **Maximum Performance**, the device power policy owner might maintain the performance level of its devices rather than reduce power consumption, and possibly leave the device powered at all times to ensure the highest level of performance.

A driver can register to be notified when a change occurs to the active power scheme, the power scheme personality, or the power source. A driver calls **PoRegisterPowerSettingCallback** to register the callback routine that the power manager calls to notify the driver of such a change, as follows:

- To register for notification of change to the active power scheme, supply the GUID that represents the setting for the power scheme (GUID_ACTIVE_POWERSCHEME). The power manager will then call the callback routine whenever the active power scheme changes, even if the personality of the new power scheme is the same as the previous power scheme.
- To register for notification of a change to the power scheme personality, supply the GUID that represents the setting for the power scheme personality (GUID_POWERSCHEME_PERSONALITY). The power manager will then call the callback routine whenever the active power scheme changes and the personality of the new power scheme is different from the personality of the previous power scheme.
- To register for notification of a change to the power source, supply the GUID that represents the setting for the power source (GUID_ACDC_POWER_SOURCE). The power manager will then call the callback routine whenever the power source setting changes.

When the active power scheme changes or the power scheme personality changes, the power manager calls the callback routine and passes the GUID that represents the new power scheme or power scheme personality. Drivers can then take the action that is appropriate for the change.

The active power scheme setting and the power scheme personality setting use the following GUIDs to identify their respective values:

- GUID_MAX_POWER_SAVINGS, which corresponds to the **Maximum Power Savings** power scheme and its corresponding personality.
- GUID_MIN_POWER_SAVINGS, which corresponds to the **Maximum Performance** power scheme and its corresponding personality.
- GUID_TYPICAL_POWER_SAVINGS, which corresponds to the **Automatic (Balanced)** power scheme and its corresponding personality.

When the power source changes, the power manager calls the callback routine and passes the GUID that represents the power source setting and the value of the power source setting that indicates whether the computer is being powered by an AC power source, a DC power source, or a short-term DC power source.

Distinguishing Fast Startup from Wake-from-Hibernation

6/25/2019 • 3 minutes to read • Edit Online

Starting with Windows 8, a fast startup mode is available to start a computer in less time than is typically required for a traditional, cold startup. A fast startup is a hybrid combination of a cold startup and a wake-from-hibernation startup. Frequently, kernel-mode device drivers need to distinguish fast startups from wake-from-hibernation so that their devices behave as users expect. To make this distinction, drivers can use information that is available in system power IRPs.

During a cold startup, the boot loader constructs a kernel memory image by loading the sections of the Windows kernel file into memory and linking them. Next, the kernel configures core system functions, enumerates the devices attached to the computer, and loads drivers for them.

In contrast, a fast startup simply loads the hibernation file (Hiberfil.sys) into memory to restore the previously saved image of the Windows kernel and loaded drivers. A fast startup tends to take significantly less time than a cold startup.

To prepare for a fast startup, Windows performs a hybrid shutdown sequence that combines elements of a full shutdown sequence and a prepare-for-hibernation sequence. First, as in a full shutdown, Windows closes all applications and logs off all user sessions. At this stage, the system state is similar to that of a computer that has just started up—no applications are running, but the Windows kernel is loaded and the system session is running. Next, the power manager sends system power IRPs to device drivers to tell them to prepare their devices to enter hibernation. Finally, Windows saves the kernel memory image (including the loaded kernel-mode drivers) in Hiberfil.sys and shuts down the computer.

By default, Windows 8 uses a fast startup in place of a cold startup. Users can typically ignore the differences between fast and cold startups, but, to meet users' expectations, fast startups should behave the same as cold startups. In particular, the devices attached to the computer should be configured the same for a fast startup as they would be for a cold startup.

If the driver for a device configures the device differently depending on whether a cold startup or a wake-from-hibernation occurred, this driver should, after a fast startup, configure the device as though a cold startup (instead of a wake-from-hibernation) occurred. For example, the system-supplied NDIS driver disables miniport wake capabilities on a fast startup but not on a wake-from-hibernation.

To distinguish a fast startup from a wake-from-hibernation, a driver can inspect the information in the system set-power (**IRP_MN_SET_POWER**) IRP that informs the driver that the computer has entered the S0 (working) state. The driver's I/O stack location in this IRP contains a **Power** member, which is a structure that contains power-related information. Starting with Windows Vista, the **Power** member structure contains a **SystemPowerStateContext** member, which is a **SYSTEM_POWER_STATE_CONTEXT** structure that contains information about the previous system power states. This information is encoded in bit fields in the **SYSTEM_POWER_STATE_CONTEXT** structure.

Most of the bit fields in the **SYSTEM_POWER_STATE_CONTEXT** structure are reserved for system use and are opaque to drivers. However, this structure contains two bit fields, **TargetSystemState** and **EffectiveSystemState**, that can be read by drivers to determine whether a fast startup or a wake-from-hibernation occurred.

The **TargetSystemState** and **EffectiveSystemState** bit fields are set to **SYSTEM_POWER_STATE** enumeration values. If **TargetSystemState** = **PowerSystemHibernate** and **EffectiveSystemState** = **PowerSystemHibernate**, a wake-from-hibernation occurred. However, if **TargetSystemState** =

PowerSystemHibernate and **EffectiveSystemState** = **PowerSystemShutdown**, a fast startup occurred.

The **TargetSystemState** bit field specifies the last system power state transition for which the driver received a system power IRP before the computer shut down or entered hibernation. The **EffectiveSystemState** bit field indicates the effective previous system power state of the device, as perceived by the user. The **TargetSystemState** and **EffectiveSystemState** values might not match if, for example, the driver received notification of a pending system transition to the hibernation state, but a hybrid shutdown subsequently occurred.

For more information, see **SYSTEM_POWER_STATE_CONTEXT**.

Calling IoCallDriver versus Calling PoCallDriver

6/25/2019 • 2 minutes to read • Edit Online

Beginning with Windows Vista, a driver should call **IoCallDriver** instead of **PoCallDriver**, to pass power IRPs to the next-lower driver. In Windows Server 2003, Windows XP, and Windows 2000, a driver must call **PoCallDriver**, not **IoCallDriver**, to pass power IRPs to the next-lower driver. Note, however, that drivers that use the same code to run both in Windows Vista and in earlier Windows versions, must call **PoCallDriver**, not **IoCallDriver**.

Beginning with Windows Vista, **PoRequestPowerIrp** and **IoCallDriver** ensure that the power manager properly synchronizes power IRPs throughout the system. In Windows Server 2003, Windows XP, and Windows 2000, **PoRequestPowerIrp**, **PoCallDriver**, and **PoStartNextPowerIrp**, ensure that the power manager properly synchronizes power IRPs throughout the system.

The system limits the number of active power IRPs as follows:

- No more than one system power IRP (**IRP_MN_SET_POWER**) at any given time.
- No more than one device set-power IRP (**IRP_MN_SET_POWER)** can be active for each PDO at any given time.
- No more than one device power IRP that requires an inrush of power can be active anywhere in the system at any given time.

To ensure that two inrush devices do not attempt to power up simultaneously, the power manager keeps track of active inrush power IRPs across the whole system and allows only one to be active at a time. An additional inrush IRP cannot start until the active inrush IRP has completed.

Because of these restrictions on inrush IRPs, a device power IRP might block while an inrush IRP for another device completes. Driver writers should be aware of this behavior while debugging.

Calling PoStartNextPowerIrp

6/25/2019 • 2 minutes to read • Edit Online

Beginning with Windows Vista, calling **PoStartNextPowerIrp** is not required and a call to this routine performs no power management operation. However, in Windows Server 2003, Windows XP, and Windows 2000, after a driver processes a query-power IRP or a set-power IRP, the driver must call **PoStartNextPowerIrp** to notify the power manager that it is ready to receive another power IRP. Drivers must call **PoStartNextPowerIrp** while the IRP stack location points to the current driver and before calling **PoCallDriver**.

A driver must call this routine once for each **IRP_MN_QUERY_POWER** or **IRP_MN_SET_POWER** request that it receives. Drivers do not need to call **PoStartNextPowerIrp** when handling **IRP_MN_WAIT_WAKE** or **IRP_MN_POWER_SEQUENCE** requests.

When a driver calls **PoStartNextPowerIrp**, the current IRP stack location must point to the current driver. As a general rule, this call is best made from an *IoCompletion* routine. **PoStartNextPowerIrp** must be called before **IoCompleteRequest**, **IoSkipCurrentIrpStackLocation**, and **PoCallDriver**. Calling the routines in the other order might cause a system deadlock.

Even if a driver fails the IRP, it must nevertheless call **PoStartNextPowerIrp** to inform the power manager that it is ready to handle another power IRP.

The following sections clarify when each type of driver should call this routine:

Calling PoStartNextPowerIrp from a Filter Driver

Calling PoStartNextPowerIrp from a Device Power Policy Owner

Calling PoStartNextPowerIrp from a Bus Driver

Calling PoStartNextPowerIrp from a Filter Driver

6/25/2019 • 2 minutes to read • Edit Online

Beginning with Windows Vista, calling **PoStartNextPowerIrp** is not required and call to this routine performs no power management operation. However, in Windows Server 2003, Windows XP, and Windows 2000, a filter driver must call **PoStartNextPowerIrp** once for every **IRP_MN_QUERY_POWER** or **IRP_MN_SET_POWER** request that the driver receives. When the call occurs depends on the type of request and whether the driver will fail or succeed the request, as the following table shows.

TYPE OF REQUEST	IF DRIVER SUCCEEDS THE REQUEST, THE CALL OCCURS:	IF DRIVER FAILS THE REQUEST, THE CALL OCCURS:
IRP_MN_QUERY_POWER (device power state)	In an *IoCompletion* routine, immediately before returning.	In *DispatchPower* routine, before calling **IoCompleteRequest**.
IRP_MN_QUERY_POWER (system power state)	In *DispatchPower* routine, after acquiring remove lock and before setting IRP stack location.	In *DispatchPower* routine, before calling **IoCompleteRequest**.
IRP_MN_SET_POWER (device power state)	In an *IoCompletion* routine, immediately before returning.	Not allowed.
IRP_MN_SET_POWER (system power state)	In *DispatchPower* routine, after acquiring remove lock and before setting IRP stack location.	Not allowed.

Calling PoStartNextPowerIrp from a Device Power Policy Owner

6/25/2019 • 2 minutes to read • Edit Online

Beginning with Windows Vista, calling **PoStartNextPowerIrp** is not required and call to this routine performs no power management operation. However, in Windows Server 2003, Windows XP, and Windows 2000, a function driver that owns device power policy must call **PoStartNextPowerIrp** once for every **IRP_MN_QUERY_POWER** or **IRP_MN_SET_POWER** request that the driver receives. When the call occurs depends on the type of request and whether the driver will fail or succeed the request, as the following table shows.

TYPE OF REQUEST	IF DRIVER SUCCEEDS THE REQUEST, THE CALL OCCURS:	IF DRIVER FAILS THE REQUEST, THE CALL OCCURS:
IRP_MN_QUERY_POWER (device power state)	In an *IoCompletion* routine, immediately before returning.	In *DispatchPower* routine, before calling **IoCompleteRequest**.
IRP_MN_QUERY_POWER (system power state)	In the **PoRequestPowerIrp** callback routine for the related device IRP, immediately before completing the system IRP.	In *DispatchPower* routine, before calling **IoCompleteRequest**.
IRP_MN_SET_POWER (device power state)	In an *IoCompletion* routine, immediately before returning.	Not allowed.
IRP_MN_SET_POWER (system power state)	In the **PoRequestPowerIrp** callback routine for the related device IRP, immediately before completing the system IRP.	Not allowed.

Calling PoStartNextPowerIrp from a Bus Driver

6/25/2019 • 2 minutes to read • Edit Online

Beginning with Windows Vista, calling **PoStartNextPowerIrp** is not required and call to this routine performs no power management operation. However, in Windows Server 2003, Windows XP, and Windows 2000, a bus driver must call **PoStartNextPowerIrp** once for every **IRP_MN_QUERY_POWER** or **IRP_MN_SET_POWER** request that the driver receives.

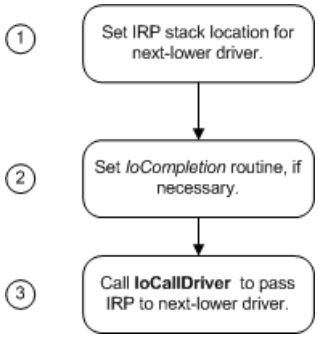
A bus driver always calls this routine in its *DispatchPower* routine, before it calls the **IoCompleteRequest** routine.

Passing Power IRPs

6/25/2019 • 4 minutes to read • Edit Online

Power IRPs must be passed all the way down the device stack to the PDO to ensure that power transitions are managed cleanly. Drivers handle an IRP that reduces device power as the IRP travels down the device stack. Drivers handle an IRP that applies device power in *IoCompletion* routines as the IRP travels back up the device stack.

The following figure shows the steps that drivers need to take to pass a power IRP down a device stack in Windows 7 and Windows Vista.



As the previous figure shows, in Windows 7 and Windows Vista, a driver must do the following:

1. Call **IoCopyCurrentIrpStackLocationToNext** if setting an *IoCompletion* routine, or **IoSkipCurrentIrpStackLocation** if not setting an *IoCompletion* routine.

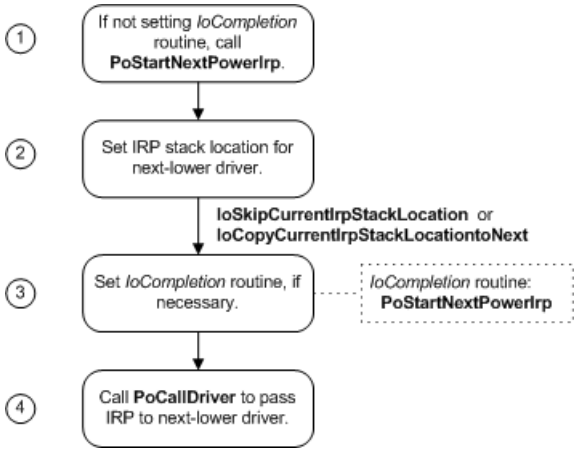
   These two routines set the IRP stack location for the next-lower driver. Copying the current stack location ensures that the IRP stack pointer is set to the correct location when the *IoCompletion* routine runs.

   If a badly written driver makes the mistake of calling **IoSkipCurrentIrpStackLocation** and then setting a completion routine, this driver might overwrite a completion routine set by the driver below it.

2. Call **IoSetCompletionRoutine** to set an *IoCompletion* routine, if a complete routine is required.

3. Call **IoCallDriver** to pass the IRP to the next-lower driver in the stack.

The following figure shows the steps that drivers need to take to pass a power IRP down a device stack in Windows Server 2003, Windows XP, and Windows 2000.



As the previous figure shows, a driver must do the following:

1. Depending on the type of driver, possibly call **PoStartNextPowerIrp**. For more information, see Calling

PoStartNextPowerIrp.

2. Call **IoCopyCurrentIrpStackLocationToNext** if setting an *IoCompletion* routine, or **IoSkipCurrentIrpStackLocation** if not setting an *IoCompletion* routine.

   These two routines set the IRP stack location for the next-lower driver. Copying the current stack location ensures that the IRP stack pointer is set to the correct location when the *IoCompletion* routine runs.

3. Call **IoSetCompletionRoutine** to set an *IoCompletion* routine. In the *IoCompletion* routine, most drivers call PoStartNextPowerIrp to indicate that it is ready to handle the next power IRP.

4. Call **PoCallDriver** to pass the IRP to the next-lower driver in the stack.

   Drivers must use **PoCallDriver**, rather than **IoCallDriver** (as for other IRPs) to ensure that the system synchronizes power IRPs properly. For more information, see Calling IoCallDriver vs. Calling PoCallDriver.

Remember that *IoCompletion* routines can be called at IRQL = DISPATCH_LEVEL. Therefore, if a driver requires additional processing at IRQL = PASSIVE_LEVEL after lower-level drivers have finished with the IRP, the driver's completion routine should queue a work item and then return STATUS_MORE_PROCESSING_REQUIRED. The worker thread must complete the IRP.

In Windows 98/Me, drivers must complete power IRPs at IRQL = PASSIVE_LEVEL.

Do Not Change the Function Codes in a Power IRP

In addition to the usual rules that govern the processing of IRPs, **IRP_MJ_POWER** IRPs have the following special requirement: A driver that receives a power IRP must not change the major and minor function codes in any I/O stack locations in the IRP that have been set by the power manager or by higher-level drivers. The power manager relies on these function codes remaining unchanged until the IRP is completed. Violations of this rule can cause problems that are difficult to debug. For example, the operating system might stop responding, or "hang."

Do Not Block While Handling a Power IRP

Drivers must not cause long delays while handling power IRPs.

When passing down a power IRP, a driver should return from its *DispatchPower* routine as soon as possible after calling **IoCallDriver** (in Windows 7 and Windows Vista) or **PoCallDriver** (in Windows Server 2003, Windows XP, and Windows 2000). A driver must not wait for a kernel event or otherwise delay before returning. If a driver cannot handle a power IRP in a brief time, it should return STATUS_PENDING and queue all incoming IRPs until the power IRP completes. (Note that this behavior is different from that of PnP IRPs and *DispatchPnP* routines, which are allowed to block.)

If the driver must wait for a power action by another driver further down the device stack, it should return STATUS_PENDING from its *DispatchPower* routine and set an *IoCompletion* routine for the power IRP. The driver can perform whatever tasks it requires in the *IoCompletion* routine, and then call **PoStartNextPowerIrp** (Windows Server 2003, Windows XP, and Windows 2000 only) and **IoCompleteRequest**.

For example, the power policy owner for a device typically sends a device power IRP while holding a system power IRP in order to set the device power state appropriate for the requested system power state.

In this situation, the power policy owner should set an *IoCompletion* routine in the system power IRP, pass the system power IRP to the next-lower driver, and return STATUS_PENDING from its *DispatchPower* routine.

In the *IoCompletion* routine, it calls **PoRequestPowerIrp** to send the device power IRP, passing a pointer to a callback routine in the request. The *IoCompletion* routine should return STATUS_MORE_PROCESSING_REQUIRED.

Finally, the driver passes down the system IRP from the callback routine. The driver must not wait for a kernel event in its *DispatchPower* routine and signal with the *IoCompletion* routine for the IRP it is currently handling; a

system deadlock might occur. For more information, see Handling a System Set-Power IRP in a Device Power Policy Owner.

In a similar situation, when the system is going to sleep, a power policy owner might need to complete some pending I/O before it sends the device IRP to power down its device. Instead of signaling an event when the I/O completes and waiting in its *DispatchPower* routine, the driver should queue a work item and return STATUS_PENDING from the *DispatchPower* routine. In the worker thread, it waits for I/O to complete and then sends the device power IRP. For more information, see **IoAllocateWorkItem**.

Queuing I/O Requests While a Device Is Sleeping

6/25/2019 • 2 minutes to read • Edit Online

While a device is asleep, its drivers should queue any I/O requests directed to the device. The **IoAllocateWorkItem**, **IoQueueWorkItem**, and **IoFreeWorkItem** support routines provide one way of queuing IRPs for delayed processing. For an example, see the queuing mechanism described for PnP drivers in Holding Incoming IRPs When A Device Is Paused.

A driver can access its device only when the device is in the Working (D0) state. A driver cannot touch any device registers when the device is in a sleep state; the device must first be returned to the Working state.

Handling Unsupported or Unrecognized Power IRPs

6/25/2019 • 2 minutes to read • Edit Online

If a driver does not support a particular power IRP, it must nevertheless pass the IRP down the device stack to the next-lower driver. A driver further down the stack might be prepared to handle the IRP and must have the opportunity to do so.

To pass an unsupported or unrecognized power IRP, a driver should call the following routines in the sequence that is described in Passing Power IRPs:

- In Windows 7 and Windows Vista, call **IoSkipCurrentIrpStackLocation** and **IoCallDriver**.
- In Windows Server 2003, Windows XP, and Windows 2000, call **PoStartNextPowerIrp**, **IoSkipCurrentIrpStackLocation**, and **PoCallDriver**.

The driver should not change anything in the IRP before passing the IRP down a device stack.

Calling ExSetTimerResolution While Processing a Power IRP

6/25/2019 • 2 minutes to read • Edit Online

During the processing of an **IRP_MJ_POWER** request, the power manager holds a lock on a resource that **ExSetTimerResolution** must acquire to complete. Consequently, a deadlock will occur if a driver directly or indirectly calls this routine while processing a power request, and then waits for the call to the routine to return before the driver completes the power request. While processing a power request, a driver can safely call **ExSetTimerResolution** only if the driver does not wait for the call to this routine to return before completing the power request. For example, a driver can create a worker thread that calls **ExSetTimerResolution**, as long as the driver then completes the power request without waiting for the call to this routine to return.

Device Power States

8/23/2019 • 4 minutes to read • Edit Online

A device power state describes the power state of a device in a computer, independently of the other devices in the computer. Device power states are named D0, D1, D2, and D3. D0 is the fully on state, and D1, D2, and D3 are low-power states. The state number is inversely related to power consumption: higher numbered states use less power. Starting with Windows 8, the D3 state is divided into two substates, D3hot and D3cold.

Device power states are characterized by the following attributes:

- Power consumption: How much power does the device use?
- Device context: How much of its operational context does the device retain in this state?
- Device driver behavior: What must the drivers for the device do to restore the device to the fully operational state?
- Restore time: How long does it take to restore the device to the fully operational state? Most types of devices have modest restore times that differ little from one device class to the next. Only a few types of devices, such as GPUs, have very large hardware contexts that take significantly longer to restore.
- Wake-up capability: Can the device request wake-up from this state? In general, if a device can request wake-up from a given power state (for example, D2), it can also request wake-up from any higher-powered state (D1).

The exact definitions of the power states are device-specific. Not all devices define all the states; many devices define only the D0 and D3 states. See the Device Class Power Management Reference Specification to find out which device power states are defined for a specific device and what the operational requirements are for each state. (The reference specifications are available at the ACPI / Power Management website.)

The power state of a device need not match the system power state. For example, some devices can be in the off (D3) state even though the system is in the system working state (S0).

The power state of a device might seem to be unrelated to the power state of the device's parent bus. For example, a USB device might be in the D2 (selective suspend) state when its parent host controller is in the D3 state. These two states appear to be inconsistent only because the definitions of the Dx states are different on USB and on the bus (typically PCI or PCI Express) that the USB host controller is connected to.

Note that some devices are capable of several different low power modes within a single device power state. Such a device can use these modes if its driver can automatically switch the device from one mode to another without changing the device power state. As a general rule, however, if there is no user-perceptible difference between the modes, the device should use only the lowest power mode. If a low power mode, such as a low-speed mode, adversely affects performance or is not transparent to software other than the device driver, the hardware should not automatically use it. See the Device Class Power Management Reference Specification for details.

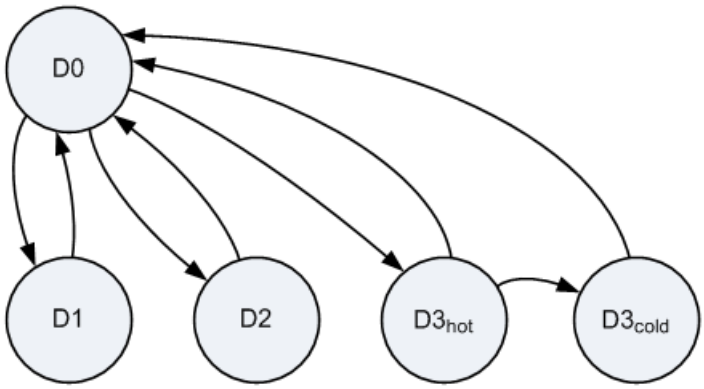
A driver or the power manager can request a device power state transition, and all drivers must be prepared to handle IRPs that request such transitions. For more information, see the following topics:

Sending IRP_MN_QUERY_POWER or IRP_MN_SET_POWER for Device Power States

Handling IRP_MN_QUERY_POWER for Device Power States

Handling IRP_MN_SET_POWER for Device Power States

Like the system, a device can transition from the working state (D0) to any low-power state (D1, D2, or D3) and from any low-power state to the working state. The following diagram is a state graph that shows the valid device power state transitions.



This graph shows the subdivision of D3 into D3hot and D3cold. D3hot and D3cold are defined starting with Windows 8. All devices are required to support the D0 state and D3hot substate. The other states shown in the diagram are optional.

In the preceding graph, the transition from D3hot to D3cold is the only direct transition between device low-power states. All other transitions between low-power states require an intermediate transition to D0, which allows the device driver to configure the device hardware, as required, either to enter the next low-power state or to stay in D0. However, a device exits D3hot and enters D3cold when power to the device is shut off, which requires no intervention from the device driver. This driver does any necessary configuration of the device hardware before the device enters D3hot; no additional configuration is required to prepare the device for the transition from D3hot to D3cold. For more information, see Supporting D3cold in a Driver.

PCI Root Port to endpoint D-state mapping

On Windows 10 systems, the overall platform power state depends on the power states (D-states) of SoC (System on Chip) integrated devices, including the PCI Root Ports. Depending on the platform being developed, the D-state requirements for PCI Root Ports may vary for each platform power state. OEMs are encouraged to refer to the IHV platform-specific documentation for platform and device power state requirements.

The table below enumerates the power state mapping of PCI Root Ports and its attached endpoints. The D-states of endpoints listed below must be achieved in order for the Root Port to enter the target D-state.

ROOT PORT TARGET D-STATE	ENDPOINT D-STATE
D0	D0, D0:F1
D0:F1	D3hot
D3hot	D3cold*

*PCI D3cold power state requires BIOS and device driver support. If support is missing, the PCI endpoint will only be able to achieve D3Hot. For more information, see Supporting D3Cold in a driver.

Device Working State D0

6/25/2019 • 2 minutes to read • Edit Online

In the D0 device power state, the device is fully on and operational. In this state, a device driver can interact with the device to perform I/O operations, and the device can generate interrupts. If the device has hardware registers that are mapped into memory or I/O address space, the driver can access these registers.

Starting with Windows 8, a device driver can connect a passive-level interrupt service routine (ISR) to the interrupt from a device. The device can generate interrupts regardless of whether it is in D0. When in a low-power Dx state, the device can generate an interrupt that acts as a trigger to bring the device back to D0. The ISR is scheduled to run at IRQL = PASSIVE_LEVEL after the device enters D0. In earlier versions of Windows, including Windows 7, a device must not generate interrupts when it is in a device power state other than D0.

A transition from D0 to a low-power Dx state can occur only when the device driver, while acting as the power policy owner for the device, initiates the transition by calling the **PoRequestPowerIrp** routine. When the power manager responds to this call by sending a power IRP (**IRP_MN_SET_POWER**), the device driver, the bus driver, and the platform firmware (through the Windows ACPI driver, Acpi.sys) cooperatively handle this IRP to change the power state of the device.

Device hardware typically monitors a set of internal events that can generate either run-time interrupts or wake signals, depending on how the device is configured. The driver implements one code path to respond to interrupts, and another to respond to wake events. The driver code can be simplified if the interrupt code path does not need to deal with wake events, and the wake code path does not need to deal with interrupts. As a best practice, the driver should configure the device to generate interrupts only when the device is in D0, and to generate wake signals only when the device is in a low-power Dx state. Typically, the driver configures the device to generate a wake signal just before the device exits D0, and configures the device to generate interrupts just after the device enters D0.

Typically, a device enters the D0 state when its hardware reset signal is asserted. In fact, the specifications for buses such as PCI and PCI Express require this behavior.

These are the characteristics of the D0 state:

Power consumption
Highest level of continuous power consumption for the device.

Device context
All context retained.

Device driver behavior
Normal operation.

Restore time
Not applicable.

Wake-up capability
Not applicable.

Device Low-Power States

6/25/2019 • 10 minutes to read • Edit Online

Device power states D1, D2, and D3 are the device low-power states. Starting with Windows 8, D3 is divided into two substates, D3hot and D3cold.

D1 and D2 are intermediate low-power states. Many classes of devices do not define these states. All devices must define D3hot.

The following sections describe D1, D2, and D3:

- Device Power State D1
- Device Power State D2
- Device Power State D3

Device Power State D1

Device power state D1 is the highest-powered device low-power state. It has the following characteristics:

Power consumption
Consumption is less than in the D0 state but greater than or equal to that in the D2 state. Frequently, D1 is a clock-gated state in which the device receives just enough power to preserve the device's hardware context. Typically, the specification for a bus or device class that supports D1 describes this state in more detail.

Device context
In general, device context is preserved by the hardware and need not be restored by the driver. The specification for a bus or device class that supports D1 typically provides detailed requirements for preserving this context.

Device driver behavior
Drivers must save and restore or reinitialize any context lost by the hardware. Typically, however, devices lose little context upon entering this state.

Restore time
In general, the time required to restore the device to D0 from D1 should be less than restoration from D2 to D0.

Wake-up capability
A device in D1 might be able to request wake-up. To supply information about whether this state can support a wake signal, a bus driver uses the **DEVICE_CAPABILITIES** structure or, starting with Windows 8, the GUID_D3COLD_SUPPORT_INTERFACE driver interface.

Typically, devices that use D1 do so because resuming from this state does not require the driver to restore the device's full hardware context. To minimize the user's perception of delay, restoring a device to D0 from D1 should incur the least possible delay. Minimizing delay in the state transition is more important than reducing power consumption.

Device Power State D2

D2 is an intermediate device low-power state with the following characteristics:

Power consumption
Consumption is less than or equal to that in the D1 state.

Device context
In general, most device context is lost by the hardware. Frequently, this state preserves the part of the context that is used to signal wake events. The specification for a bus or device class that supports D2 typically provides

detailed requirements for preserving this context.

Device driver behavior
Device drivers must save and restore or reinitialize any context lost by the hardware. A typical device loses most context when it enters D2.

Restore time
Restoring the device from D2 to D0 takes at least as long as restoring the device from D1 to D0. A graphics adapter that has a large frame buffer is an example of a device that has a large amount of hardware context to restore after a transition from D2 to D0. For such a device, the restore time from D2 might be much greater than the restore time from D1.

Wake-up capability
A device in D2 might be able to request wake-up. To supply information about whether this state can support a wake signal, a bus driver uses the **DEVICE_CAPABILITIES** structure or, starting with Windows 8, the GUID_D3COLD_SUPPORT_INTERFACE driver interface.

Typically, drivers that support D2 do so because their devices cannot support wake from D3. For these devices, power consumption in the D2 state drops to the lowest level from which the device can recover in response to a wake signal. In contrast to the D1 state, which is implemented to reduce the delay perceived by the user, the goal in implementing the D2 state is to conserve power. As a result, the restore time from D2 to D0 typically exceeds that from D1 to D0. In the D2 state, for example, reduced power on the bus might cause a device to turn off some of its functionality, thus requiring additional time to restart and restore the device.

Many classes of device do not define this state.

Device Power State D3

D3 is the lowest-powered device low-power state. All devices must support this state.

Starting with Windows 8, the operating system subdivides D3 into two separate and distinct substates, D3hot and D3cold. Earlier versions of Windows define the D3 state, but not the D3hot and D3cold substates. However, all versions of the PCI Bus Power Management Interface Specification define separate D3hot and D3cold substates, and versions 4 and later of the Advanced Configuration and Power Interface Specification define D3hot and D3cold substates.

Although versions of Windows before Windows 8 do not explicitly define the D3hot and D3cold substates of D3, these substates exist implicitly in these earlier versions of Windows. A device is implicitly in the D3hot substate if the device is explicitly in the D3 state, and the computer is in the S0 system power state. In D3hot, a device is connected to a power source (although the device might be configured to draw low current), and the presence of the device on the bus can be detected. A device is implicitly in the D3cold substate if it is explicitly in the D3 state, and the computer is in a low-power Sx state (a state other than S0). In this implicit D3cold substate, the device might receive a trickle current, but the device and the computer are effectively turned off until a wake event occurs.

Starting with Windows 8, a device can enter and leave the D3cold substate while the computer remains in the S0 state. To support this new behavior, D3hot and D3cold must be explicitly defined as distinct substates of D3.

D3hot is the only substate of D3 that the device can enter directly from D0. A device makes a transition from D0 to D3hot under software control by the device driver. In D3hot, the device can be detected on the bus that it connects to. The bus must remain in the D0 state while the device is in the D3hot substate. From D3hot, the device can either return to D0 or enter D3cold. D3cold can be entered only from D3hot.

D3cold is a substate of D3 in which the device is physically connected to the bus but the presence of the device on the bus cannot be detected (that is, until the device is turned on again). In D3cold, one or both of the following is true:

- The bus that the device connects to is in a low-power state.
- The device is in a low-power state in which the device does not respond when the bus driver tries to detect its

presence on the bus.

The transition from D3hot to D3cold occurs with no device driver interaction. Instead, the device driver indicates whether it is prepared for a D3cold transition before it initiates the transition from D0 to D3hot. Subsequently, a transition from D3hot to D3cold may or may not occur, depending on whether all of the conditions are right to enable this transition.

Two such conditions are that all of the devices that use the same power source are in D3hot and are prepared for a D3cold transition. When the last of these devices enters D3hot, the parent bus driver or ACPI filter driver turns off the power source to these devices, which is to say that the devices enter D3cold.

A device that is in D3cold can leave this substate only by entering D0. There is no direct transition from D3cold to D3hot.

When the computer is in the S0 state and a device enters the D3hot substate, the device driver is typically unable to determine in advance whether the device's next transition will be to D3cold or D0. The one exception is when the computer is preparing to leave the S0 state. In this case, the next transition is to D3cold.

The following sections describe D3hot and D3cold:

- D3hot substate
- D3cold substate

For more information, see Supporting D3cold in a Driver.

D3hot substate

D3hot has the following characteristics:

Power consumption
Power is mostly removed from the device, but not from the computer as a whole. The computer, which is in the S0 state, might continue running in this state, or it might be preparing to move from S0 to a low-power Sx state.

Device context
The device driver is solely responsible for restoring device context. The driver must preserve and then restore all device context or must reinitialize the device upon transition to the D0 state.

Device driver behavior
The device driver is solely responsible for restoring device context, typically from the most recent working configuration.

Restore time
Total restore time is the highest of any of the device power states, except for D3cold, but is typically not much greater than the restore time from D2.

Wake-up capability
A device in the D3hot substate may or may not be able to request wake-up. To supply information about whether this substate can support a wake signal, a bus driver uses the **DEVICE_CAPABILITIES** structure or, starting with Windows 8, the GUID_D3COLD_SUPPORT_INTERFACE driver interface.

In D3hot, only minimal trickle current is available. Drivers and hardware must be prepared for the absence of power. The specification for a bus that supports D3hot typically provides detailed requirements for power sources that can be used in this state. To return the device to the working state, the device's drivers must be able to restore and reinitialize the device without depending on the BIOS to run any code in the option ROM that might be available for the device.

The parent bus driver will not remove system power from the parent bus of any device that enters D3hot unless the computer as a whole transitions to the S0 state.

All classes of device define the D3hot substate.

D3cold substate

D3cold has the following characteristics:

Power consumption
Power has been fully removed from the device and possibly from the entire system. The device may be able to draw current from side-band sources, depending on its construction.

Device context
The device driver is solely responsible for restoring device context. The driver must preserve and then restore device context or must reinitialize the device upon transition to the D0 state.

Device driver behavior
The device driver is solely responsible for restoring device context, typically from the most recent working configuration.

Restore time
Total restore time is the highest of any of the device power states.

Wake-up capability
In the D3cold substate, a device might be able to trigger a wake signal to wake a sleeping computer. This capability is reported in the **DEVICE_CAPABILITIES** structure and, starting with Windows 8, by the *GetIdleWakeInfo* routine in the GUID_D3COLD_SUPPORT_INTERFACE driver interface. After the signal wakes the computer, the device driver initiates the device's transition from D3cold to D0. For more information, see the following remarks.

Starting with Windows 8, a device in the D3cold substate might be able to trigger a wake signal to a computer that is in the S0 system power state. This capability is reported by the *GetIdleWakeInfo* routine. The **DEVICE_CAPABILITIES** structure does not contain information about this capability. After the wake signal arrives, the device driver initiates the device's transition from D3cold to D0. In this case, the computer is awake when the signal arrives, and only the device needs to wake.

In many existing hardware platforms, a device that is in a low-power Dx state can trigger a wake signal to wake a sleeping computer. However, the same device might not be able to trigger a wake signal if the computer is running in the S0 state. Thus, the driver for this device must not initiate the device's transition from D0 to a low-power Dx state when the computer is in the S0 state. Otherwise, after the device leaves D0, it will be unavailable until the computer leaves the S0 state. This device should leave the D0 state only when the computer is preparing to leave the S0 state.

If a device that is in a low-power Dx state can trigger a wake signal to a computer that is running in the S0 state, the device is not required to remain in D0 when the computer is in S0. If the computer is in S0, and the device is in D0 but is idle, the driver can arm the device to trigger a wake signal, and then initiate the device's transition from D0 to this low-power Dx state.

Some classes of device define the D3cold substate.

For more information, see Supporting D3cold in a Driver.

Required Support for Device Power States

6/25/2019 • 2 minutes to read • Edit Online

Consult the relevant Device Class Power Management Reference Specification to find out which device power states are defined for the class of device with which you are working with and what the operational requirements are for each state. These specifications are available at the ACPI / Power Management website.

Legacy devices and other devices for which no power management specification exists should follow the Default Device Class Power Management Specification. The default specification requires:

- Support for the D0 and D3 states.
- A driver that saves and restores or reinitializes device context when the device is powered on.
- A driver that manages the device power policy.

Class and port drivers supplied with the system and by independent hardware vendors (IHVs) typically support power management. If you are writing a minidriver that links to such a driver, check the relevant class or port driver documentation in the Windows Driver Kit (WDK) to find out the extent of power management support required in the minidriver. The following general guidelines apply:

- A network adapter driver must conform to the Network Driver Interface Specification 6.00 (NDIS 6.0) (Windows Vista) or to NDIS 5.0 (Windows Server 2003, Windows XP, and Windows 2000). In addition, the driver must conform to the power management requirements for the device setup class of the driver and the Windows version of the driver.
- Streaming drivers use the power management interfaces in the streaming class driver to handle device power states D0 and D3. To handle device power states D1 and D2, these drivers must use the power management interfaces described in this section.
- The SCSI port driver manages most of the PnP and power management requirements for the miniport. SCSI miniport drivers must support PnP and power management interfaces along with related routines such as **HwScsiAdapterControl**.
- The video port driver manages most of the PnP and power management requirements for the miniport. Video miniport drivers must support miniport-specific routines, which are described elsewhere in the WDK.

Managing Device Power Policy

6/25/2019 • 2 minutes to read • Edit Online

Just as the power manager maintains and administers power policy for the system, one driver in the device stack for each device maintains and administers power policy for the device. This driver is the *device power policy owner* for the device.

The device power policy owner is the driver that has the most information about the device usage and power state. It need not physically be able to set the device registers to power the device on and off, but it must be able to determine when the device is in use, when it is idle, and when it should change power state.

Typically, the function driver for a device is its power policy owner, although for some devices another driver or system component might assume this role. For more information about the types of drivers involved in power management, see Types of WDM Drivers.

Some drivers act as the function driver for one device (creating an FDO) and the bus driver (creating a PDO) for an enumerated child device. In its Dispatch routines for power and PnP IRPs, such a driver must distinguish its handling of IRPs sent to the FDO and those sent to the PDO.

For example, the driver for a SCSI adapter might perform the roles of function driver (creating an FDO) for the adapter itself and bus driver (creating a PDO) for the disks attached to the adapter. In its capacity as function driver/policy owner for the SCSI adapter, this driver receives system IRPs and requests device IRPs for the SCSI adapter. In its capacity as bus driver for the disks, it handles and completes device IRPs that specify the disk PDOs it creates. Just because the driver owns power policy for one device (FDO) does not mean that it owns power policy for the child device (PDO).

The device power policy owner is responsible for the following:

- Setting the initial power state of the device to D0 by calling **PoSetPowerState** as it handles the Plug and Play manager's **IRP_MN_START_DEVICE** request.

  Devices should power on as needed; for example, a device must power on to handle an I/O request. The device power policy owner is responsible for determining when its device is needed, ensuring that device power is on, and setting the correct device power state. The typical device should be powered on by the time the PnP start-device IRP has completed.

  As a general rule, most devices should be powered off when not in use, even when the system is in the working state.

- Sending a device power request in response to a system power request by calling **PoRequestPowerIrp**.

  For example, when the policy owner receives a system set-power IRP, it sends a device set-power IRP. Most devices enter D3 when the system enters any sleeping state. The **DeviceState** array in the **DEVICE_CAPABILITIES** structure lists the highest-powered state the device can maintain for each system power state. (See Reporting Device Power Capabilities.)

- Detecting when the device is idle and putting it to sleep to conserve energy.

  Either the power manager or the device policy owner can detect an idle device and send a device power IRP to change its state. For more information, see Detecting an Idle Device.

- Returning its device to the working state when required.

  When an I/O request arrives for a sleeping device, the device's drivers should return it to the working state.

- Enabling and disabling wake-up for its device when requested.

  The device power policy owner sends and cancels wait/wake IRPs, as described in Supporting Devices that Have Wake-Up Capabilities.

Handling IRP_MN_SET_POWER for Device Power States

6/25/2019 • 3 minutes to read • Edit Online

A device set-power IRP requests a change of state for a single device and is sent to all the drivers in the stack for the device. Such an IRP specifies **DevicePowerState** in the **Power.Type** member of the I/O stack location.

Drivers handle power-down IRPs as they travel down the stack. For power-up IRPs, drivers set *IoCompletion* routines as the IRPs travel down the stack, and then handle the IRPs in the *IoCompletion* routines as the IRPs travel back up the stack. The drivers in a typical device stack handle a device set-power IRP as follows:

- Most filter drivers should simply call **IoMarkIrpPending**, pass the IRP to the next-lower driver (see Passing Power IRPs), and return STATUS_PENDING from the *DispatchPower* routine. Some filter drivers, however, might first need to perform device-specific tasks, such as queuing incoming IRPs or saving device power state.
- A function driver calls **IoMarkIrpPending**, performs device-specific tasks (such as completing pending I/O requests, queuing incoming I/O requests, saving device context, or changing device power), sets an *IoCompletion* routine if necessary, and passes the device power IRP to the next-lower driver (see Passing Power IRPs). It returns STATUS_PENDING from its *DispatchPower* routine.
- The bus driver changes device power if it is capable of doing so and then calls **PoSetPowerState** to notify the power manager of the new device power state. In Windows Server 2003, Windows XP, and Windows 2000 only, the driver must also call **PoStartNextPowerIrp** to start the next power IRP after it sets the power state. The driver then completes the IRP, specifying IO_NO_INCREMENT. If the driver cannot complete the IRP immediately, it calls **IoMarkIrpPending**, returns STATUS_PENDING from its *DispatchPower* routine, and completes the IRP later.

Even if the target device is already in the requested power state, each function or filter driver must pass the IRP down to the next-lower driver. Every set-power IRP must travel all the way down the device stack to the bus driver, which completes it.

Function and filter drivers that are located above a bus driver must not fail a device set power IRP. The bus driver can fail a device power-up IRP if the device is removed or in the process of being removed.

Each driver (function, filter, and bus driver) in a driver stack must call **PoSetPowerState** to inform the power manager of a change in the power state of its corresponding device object.

Like other driver tasks associated with device power-up and power-down, the call to **PoSetPowerState** must occur after the device powers on (if the new state is D0) or before the device powers off (if the new state is any other state).

Each driver should keep track of the power state of its device. The power manager does not supply this information to drivers.

While handling an **IRP_MN_SET_POWER** request for a device power state, a driver should return from the *DispatchPower* routine as quickly as possible. A driver must not wait in its *DispatchPower* routine for a kernel event signaled by code that handles the same IRP. Because power IRPs are synchronized throughout the system, a deadlock might occur.

To ensure the highest level of system performance, especially for multimedia applications, a driver should perform time-consuming operations at an interrupt request level (IRQL) equal to PASSIVE_LEVEL. To perform operations

at IRQL = PASSIVE_LEVEL, a driver can use a dedicated thread or a system worker thread. For guidelines on optimizing driver performance for multimedia platforms, see the Streaming Media Devices Design Guide.

The exact steps a driver must take to handle a power IRP depend upon whether the device is powering up or down, as described in the following sections:

Handling Device Power-Down IRPs

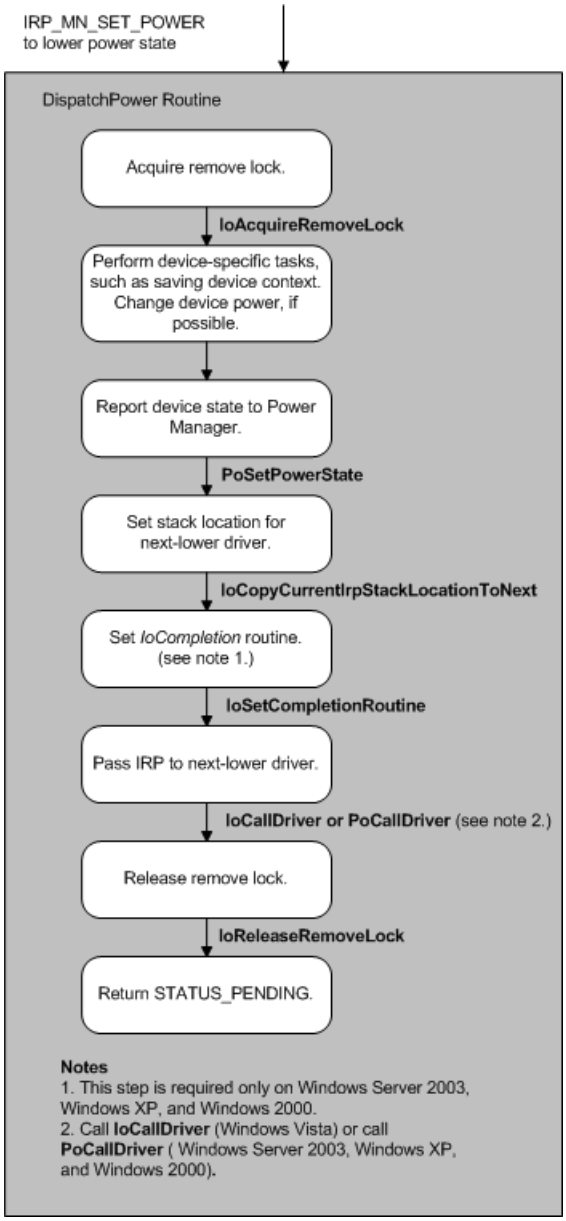
Handling Device Power-Up IRPs

Handling Device Power-Down IRPs

6/25/2019 • 4 minutes to read • Edit Online

A device power-down IRP specifies the minor function code **IRP_MN_SET_POWER** and a device power state (**PowerDeviceD0**, **PowerDeviceD1**, **PowerDeviceD2**, or **PowerDeviceD3**) that is less-powered or equal to the current device power state. Drivers must handle the power-down IRP as the IRP travels down the device stack. Higher-level drivers must handle the IRP before lower-level drivers. Drivers that have no device-specific tasks to perform should promptly pass the IRP to the next-lower driver.

The following figure shows the steps involved in handling such an IRP.



If the IRP specifies **PowerDeviceD3**, the function driver should typically perform the following tasks:

- Call **IoAcquireRemoveLock**, passing the current IRP, to ensure that the driver does not receive a PnP **IRP_MN_REMOVE_DEVICE** request while handling the power IRP.

  If **IoAcquireRemoveLock** returns a failure status, the driver should not continue processing the IRP. Instead, beginning with Windows Vista, the driver should call **IoCompleteRequest** to complete the IRP and then return the failure status. In Windows Server 2003, Windows XP, and Windows 2000, the driver

should call **IoCompleteRequest** to complete the IRP, then call **PoStartNextPowerIrp** to start the next power IRP, and then return the failure status.

- Perform any device-specific tasks that must be done before device power is removed, such as closing the device, completing or flushing any pending I/O, disabling interrupts, queuing subsequent incoming IRPs, and saving device context from which to restore or reinitialize the device.

  The driver should not cause a long delay (for example, a delay that a user might find unreasonable for this type of device) while handling the IRP.

  The driver should queue any incoming I/O requests until the device has returned to the working state.

- Possibly check the value at **Parameters.Power.ShutdownType**. If a system set-power IRP is active, the **ShutdownType** provides information about the system IRP. For more information about this value, see System Power Actions.

  Drivers of devices on the hibernate path must inspect this value. If the **ShutdownType** is **PowerActionHibernate**, the driver should save any context required to restore the device but should not power down the device.

- Change the physical power state of the device if the driver is capable of doing so and if the change is appropriate.

- Call **PoSetPowerState** to notify the power manager of the new device power state.

- Call **IoCopyCurrentIrpStackLocationToNext** to set up the stack location for the next-lower driver.

- Set an *IoCompletion* routine that calls **PoStartNextPowerIrp** that indicates that the driver is ready to handle the next power IRP. This step is not required in Windows 7 and Windows Vista.

- Call **IoCallDriver** (in Windows 7 and Windows Vista) or call **PoCallDriver** (in Windows Server 2003, Windows XP, and Windows 2000) to pass the IRP to the next-lower driver. The IRP must be passed all the way down to the bus driver, which completes the IRP.

- Call **IoReleaseRemoveLock** to release the previously acquired lock.

- Return STATUS_PENDING.

Drivers must save any device context information and set the new power state before forwarding the IRP. The context information should contain, at minimum, the requested new power state. It should also include any additional information the driver will need upon power-up. After the IRP has been completed and the device has been powered off, the driver can no longer access the device and device context is not available.

Each driver must pass the IRP to the next-lower driver. When the IRP reaches the bus driver, the bus driver powers off the device (if it is capable of this), calls **PoSetPowerState** to inform the power manager, and completes the IRP.

However, if the bus driver services the hibernation device, it should check whether the value of **ShutdownType** in the IRP is PowerSystemHibernate. If so, the bus driver should call **PoSetPowerState** to report PowerDeviceD3 but should not power down the device. The device will power down after the hibernate file is saved, along with the rest of the system.

After all of its child devices power down, a bus driver might choose to power down its bus also. Such behavior is device-dependent.

If the IRP specifies any other state (D0, D1 or D2), required driver actions are device-dependent. Typically, devices that support these states can quickly return to the working state when an I/O request arrives. A driver for such a device must complete any pending I/O requests, queue any new requests, and save all necessary context before forwarding the IRP to the next-lower driver. When the IRP reaches the bus driver, it sets the hardware in the

requested state. A driver cannot access the device while it is asleep.

Under some circumstances, a function or filter driver might receive a device power IRP specifying PowerDeviceD0 when the device is already in the D0 state. The driver should handle this IRP as it would any other set-power IRP: complete pending I/O requests, queue incoming I/O requests, set an *IoCompletion* routine, and pass the IRP down to the next-lower driver. A driver must not, however, change the device's hardware settings. When the bus driver receives the IRP, it should simply complete the IRP. When the IRP completes, function and filter drivers can handle any queued requests. Queuing I/O until the IRP completes eliminates any possibility of lower drivers attempting to change device registers while a higher driver attempts I/O.

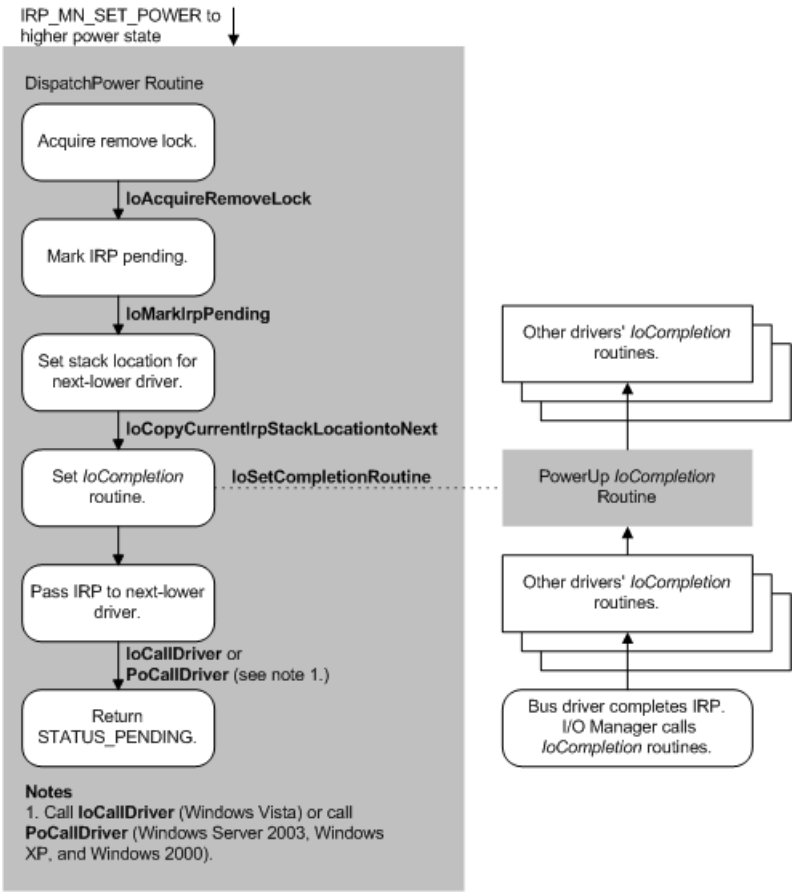
Handling Device Power-Up IRPs

6/25/2019 • 2 minutes to read • Edit Online

Device power-up IRPs specify **IRP_MN_SET_POWER** and a device power state that requires more power than the current device power state. Typically, a power-up IRP specifies the device working state **PowerDeviceD0**.

Requests to power up a device must be handled first by the underlying bus driver for the device, and then by each successive driver going back up the stack.

The following figure shows the steps involved in handling a power-up IRP.



When handling an **IRP_MN_SET_POWER** request for power-up, a function or filter driver must:

- Call **IoAcquireRemoveLock** to ensure that the driver does not receive an **IRP_MN_REMOVE_DEVICE** request while handling the power-up IRP.

  If **IoAcquireRemoveLock** returns a failure status, the driver should not continue processing the IRP. Instead, beginning with Windows Vista, the driver should call **IoCompleteRequest** to complete the IRP and then return the failure status. In Windows Server 2003, Windows XP, and Windows 2000, the driver should call **IoCompleteRequest** to complete the IRP, then call **PoStartNextPowerIrp** to start the next power IRP, and then return the failure status.

- Call **IoMarkIrpPending** to mark the IRP pending.

- Call **IoCopyCurrentIrpStackLocationToNext** to set the IRP stack location. A driver must not call **IoSkipCurrentIrpStackLocation** if it sets an *IoCompletion* routine.

- Call **IoSetCompletionRoutine** to set a power-up *IoCompletion* routine.

When handling a device power-up IRP, the driver should set an *IoCompletion* routine to restore context, release the remove lock, and perform other required tasks after the IRP is complete and the device powers on. The driver should not restore context before the IRP has completed. For more information, see IoCompletion Routines for Device Power IRPs.

- Call **IoCallDriver** (in Windows 7 and Windows Vista) or **PoCallDriver** (Windows Server 2003, Windows XP, and Windows 2000) to pass the IRP to the next-lower driver. The IRP must travel all the way down the device stack to the bus driver. Only the bus driver is allowed to complete the IRP.
- Return STATUS_PENDING.

When the bus driver receives the IRP, it should first check to ensure the device is still present and has not been removed or replaced while asleep. If the device is no longer present, the bus driver should call **IoInvalidateDeviceRelations** on the parent device to notify the Plug and Play manager that the device has disappeared. In this situation, the bus driver can fail the device power-up IRP.

If the device is still present, the bus driver then performs the tasks required to return the device to an operating condition, calls **PoSetPowerState** to inform the power manager of the new device power state, and completes the IRP (**IoCompleteRequest**). If drivers have queued I/O while the device was sleeping, or if the device requires inrush power, the bus driver applies power to the device. Otherwise, the bus driver applies power as soon as it has to communicate with the device.

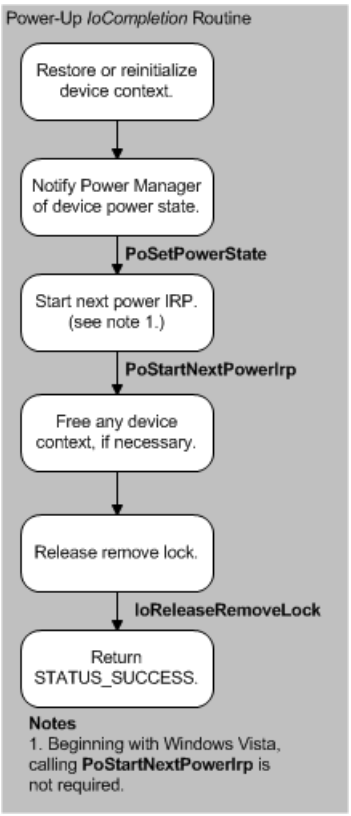
For a list of best practices to achieve fast startup times from power-off, standby, and hibernation states, see Improving System Startup Performance.

IoCompletion Routines for Device Power IRPs

6/25/2019 • 2 minutes to read • Edit Online

After the bus driver completes the IRP, the I/O manager calls the *IoCompletion* routines registered by higher-level drivers as they passed the IRP down the stack.

Whenever a device enters the D0 state, each of its drivers should set an *IoCompletion* routine that performs most of the tasks required to return it to the working state. Drivers should set an *IoCompletion* routine for any transition to the D0 state, whether the device is returning from a sleeping state or entering D0 at system start-up. The following figure shows the tasks such an *IoCompletion* routine should perform.



These tasks include:

- Restoring device power state or reinitializing the device, as required, and preparing to handle any I/O queued by drivers while the device was not in the working state
- Calling **PoSetPowerState** to notify the power manager that the device is in the D0 power state.
- Calling **PoStartNextPowerIrp** to receive the next power IRP, if the driver did not originally send the current power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only).
- Freeing memory allocated for the device context.
- Calling **IoReleaseRemoveLock** to free the lock that the driver acquired in its *DispatchPower* routine when it received the IRP.
- Returning STATUS_SUCCESS.

The bus driver does not power up the device until it or higher drivers must communicate with the device.

When its device enters a sleeping state, a driver should set an *IoCompletion* routine that calls **PoStartNextPowerIrp** (Windows Server 2003, Windows XP, and Windows 2000 only) and releases the remove

lock. Remember that a driver cannot access its device while the device is in a sleeping state.

Handling IRP_MN_QUERY_POWER for Device Power States

6/25/2019 • 3 minutes to read • Edit Online

A device query-power IRP queries about a change of state for a single device and is sent to all of the drivers in the stack for the device. Such an IRP specifies **DevicePowerState** in the **Power.Type** member of the I/O stack location.

Drivers handle query-power IRPs as they travel down the stack.

A function or filter driver can fail an **IRP_MN_QUERY_POWER** request if any of the following is true:

- The device is enabled for wake-up and the requested power state is below the state from which the device can wake the system. For example, a device that can wake the system from D2 but not from D3 would fail a query for D3 but succeed a query for D2.
- Entering the requested state would force the driver to abandon an operation that would lose data, such as an open modem connection. A driver rarely will fail a query for this reason; under most circumstances, the application handles such cases.

To fail an **IRP_MN_QUERY_POWER** request, a driver takes the following steps:

1. Call **PoStartNextPowerIrp** to indicate that the driver is prepared to handle the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.)
2. Set **Irp->IoStatus.Status** to a failure status and call **IoCompleteRequest**, specifying IO_NO_INCREMENT. The driver does not pass the IRP further down the device stack.
3. Return an error status from its *DispatchPower* routine.

If the driver succeeds the query-power IRP, it must not start any operations or take any other action that would prevent its successful completion of a subsequent **IRP_MN_SET_POWER** request to the queried power state.

A driver that succeeds the IRP must prepare for a set-power IRP for the queried state and pass down the query IRP, as follows:

1. Finish any outstanding I/O operations.
2. Queue incoming I/O requests.
3. Avoid starting any other new activities that would interfere with a transition to the specified power state. However, the driver should not save device context or take other steps toward shutdown.
4. Call **IoCopyCurrentIrpStackLocationToNext** to set the IRP stack location for the next-lower driver.
5. Set an *IoCompletion* routine. In the *IoCompletion* routine, call **PoStartNextPowerIrp** (Windows Server 2003, Windows XP, and Windows 2000 only) to indicate the driver's readiness to handle the next power IRP.
6. Call **IoCallDriver** (in Windows 7 and Windows Vista) or **PoCallDriver** (in Windows Server 2003, Windows XP, and Windows 2000) to pass the query IRP to the next-lower driver. Do not complete the IRP.
7. Return STATUS_PENDING. The driver must not change the value at **Irp->IoStatus.Status**.

When the query-power IRP reaches the bus driver, the bus driver calls **PoStartNextPowerIrp** (Windows Server

2003, Windows XP, and Windows 2000 only) and sets **Irp->IoStatus.Status** to STATUS_SUCCESS if the driver can change to the specified power state or sets a failure status if it cannot. The bus driver then calls **IoCompleteRequest**, specifying IO_NO_INCREMENT.

The drivers in a typical device stack handle a device query-power IRP as follows:

- Most filter drivers should simply pass the IRP to the next-lower driver (see Passing Power IRPs) and return STATUS_PENDING. Some filter drivers, however, might first need to perform device-specific tasks, such as queuing incoming IRPs or saving device power state.
- A function driver performs device-specific tasks (such as, completing pending I/O requests, queuing incoming I/O requests, saving device context, or changing device power), sets an *IoCompletion* routine, and passes the device power IRP to the next-lower driver (see Passing Power IRPs). It returns STATUS_PENDING from its *DispatchPower* routine.
- The bus driver calls **PoStartNextPowerIrp** (Windows Server 2003, Windows XP, and Windows 2000 only) to start the next power IRP. It then completes the IRP, specifying IO_NO_INCREMENT. If the driver cannot complete the IRP immediately, it calls **IoMarkIrpPending**, returns STATUS_PENDING from its *DispatchPower* routine, and completes the IRP later.

Even if the target device is already in the queried power state, each function or filter driver must queue I/O and pass the IRP down to the next-lower driver. The IRP must travel all the way down the device stack to the bus driver, which completes it.

While handling an **IRP_MN_QUERY_POWER** request, a driver should return from the *DispatchPower* routine as quickly as possible. A driver must not wait in its *DispatchPower* routine for a kernel event signaled by code that handles the same IRP. Because power IRPs are synchronized throughout the system, a deadlock might occur.

Sending IRP_MN_QUERY_POWER or IRP_MN_SET_POWER for Device Power States

6/25/2019 • 4 minutes to read • Edit Online

A device power policy owner sends a device query-power IRP (**IRP_MN_QUERY_POWER**) to determine whether lower drivers can accommodate a change in device power state, and a device set-power IRP (**IRP_MN_SET_POWER**) to change the device power state. (This driver can also send a wait/wake IRP to enable its device to awaken in response to an external signal; see Supporting Devices that Have Wake-Up Capabilities for details.)

The driver should send an **IRP_MN_QUERY_POWER** request when either of the following is true:

- The driver receives a system query-power IRP.
- The driver is preparing to put an idle device in a sleep state, so must query lower drivers to find out whether doing so is feasible.

The driver should send an **IRP_MN_SET_POWER** request when any of the following is true:

- The driver has determined that the device is idle and can be put to sleep.
- The device is sleeping and must re-enter the working state to handle waiting I/O.
- The driver receives a system set-power IRP.

A driver must not allocate its own power IRP; the power manager provides the **PoRequestPowerIrp** routine for this purpose. As Rules for Handling Power IRPs explains, **PoRequestPowerIrp** allocates and sends the IRP, and in combination with **IoCallDriver** (in Windows 7 and Windows Vista), or **PoCallDriver** (in Windows Server 2003, Windows XP, and Windows 2000), ensures that all power requests are properly synchronized. Callers of **PoRequestPowerIrp** must be running at IRQL <= DISPATCH_LEVEL.

The following is the prototype for this routine:

```
NTSTATUS
PoRequestPowerIrp (
    IN PDEVICE_OBJECT DeviceObject,
    IN UCHAR MinorFunction,
    IN POWER_STATE PowerState,
    IN PREQUEST_POWER_COMPLETE CompletionFunction,
    IN PVOID Context,
    OUT PIRP *Irp OPTIONAL
    );
```

To send the IRP, the driver calls **PoRequestPowerIrp**, specifying a pointer to the target device object in *DeviceObject*, the minor IRP code IRP_MN_SET_POWER or IRP_MN_QUERY_POWER in *MinorFunction*, the value **DevicePowerState** in the *PowerState*.**Type**, and a device power state in *PowerState*.**State**. In Windows 98/Me, *DeviceObject* must specify the PDO of the underlying device; in Windows 2000 and later versions of Windows, this value can point to either the PDO or an FDO of a driver in the same device stack.

If the driver must perform additional tasks after all other drivers have completed the IRP, it should pass a pointer to a power completion function in *CompletionFunction*. The I/O manager calls the *CompletionFunction* after calling all the *IoCompletion* routines set by drivers as they passed the IRP down the stack.

Whenever a device power policy owner sends a device power query IRP, it should subsequently send a device set-

power IRP from the callback routine (*CompletionFunction*) that it specified in the call to **PoRequestPowerIrp**. If the query succeeded, the set-power IRP specifies the queried power state. If the query failed, the set-power IRP re-asserts the current device power state. Re-asserting the current state is important because drivers queue I/O in response to the query; the policy owner must send the set-power IRP to notify drivers in its device stack to begin processing queued I/O requests.

Keep in mind that the policy owner for a device not only sends the device power IRP but also handles the IRP as it is passed down the device stack. Therefore, such a driver often sets an *IoCompletion* routine (with **IoSetCompletionRoutine**) as part of its IRP-handling code, particularly when the device is powering up. The *IoCompletion* routine is called in sequence with *IoCompletion* routines set by other drivers and before the *CompletionFunction*. For further information, see IoCompletion Routines for Device Power IRPs.

Because the IRP has been completed by all drivers when the *CompletionFunction* is called, the *CompletionFunction* must not call **IoCallDriver**, **PoCallDriver**, or **PoStartNextPowerIrp** with the IRP it originated. (It might, however, call these routines for a different power IRP.) Instead, this routine performs any additional actions required by the driver that originated the IRP. If the driver sent the device IRP in response to a system IRP, the *CompletionFunction* might complete the system IRP. For further information, see Handling a System Set-Power IRP in a Device Power Policy Owner.

In response to the call to **PoRequestPowerIrp**, the power manager allocates a power IRP and sends it to the top of the device stack for the device. The power manager returns a pointer to the allocated IRP.

If no errors occur, **PoRequestPowerIrp** returns STATUS_PENDING. This status means that the IRP has been sent successfully and is pending completion. The call fails if the power manager cannot allocate the IRP or if the caller has specified an invalid minor power IRP code.

Requests to power up a device must be handled first by the underlying bus driver for the device and then by each successively higher driver in the stack. Therefore, when sending a **PowerDeviceD0** request, the driver must ensure that its *CompletionFunction* performs required tasks after the IRP is complete and the device is powered on.

When powering off a device (**PowerDeviceD3**), each driver in the device stack must save all of its necessary context and do any necessary clean-up before sending the IRP to the next-lower driver. The extent of the context information and clean-up depends on the type of driver. A function driver must save hardware context; a filter driver might need to save its own software context. A *CompletionFunction* set in this situation can take actions associated with a completed power IRP, but the driver cannot access the device.

Detecting an Idle Device

12/5/2018 • 2 minutes to read • Edit Online

The power policy owner for each device is responsible for determining when the device is idle and can be put to sleep to conserve power. The policy owner has two options for detecting an idle device:

Use Power Manager routines for idle detection

Perform device-specific idle detection

Using Power Manager Routines for Idle Detection

6/25/2019 • 2 minutes to read • Edit Online

The power manager provides support for idle detection through the **PoRegisterDeviceForIdleDetection** and **PoSetDeviceBusy** routines.

To enable idle detection for its device, a device power policy owner calls **PoRegisterDeviceForIdleDetection** and specifies:

- The idle time-out value to apply when the system optimizes for performance.
- The idle time-out value to apply when the system optimizes for conservation.
- The device power state to which the device should transition when idle.

PoRegisterDeviceForIdleDetection returns a pointer to an idle counter, which the driver uses later to disable idle detection. Callers of **PoRegisterDeviceForIdleDetection** must be running at IRQL < DISPATCH_LEVEL.

A driver can register its device for idle detection any time after the device has been started and is ready to handle device power IRPs. For example, a driver might enable idle detection as part of its *IoCompletion* routine for a PnP start-device IRP.

The time-out values for any given device should be proportional to the device's power-up latency and based on observed device behavior. For devices of certain types, a driver can specify conservation and performance time-out values of -1 to use the standard power policy time-outs for the device class. See the device-specific documentation for details.

When the device is in use, the driver must call **PoSetDeviceBusy**, passing the pointer returned by **PoRegisterDeviceForIdleDetection**. **PoSetDeviceBusy** resets the idle counter, thus restarting the idle countdown for the device. The driver should call **PoSetDeviceBusy** on every I/O operation.

To determine whether the device is idle, the power manager compares the value of the idle counter with the driver-specified idle time-out value for the current system power policy (either conservation or performance). See the Microsoft Windows SDK for functions pertaining to the system power policy.

When the device satisfies the time-out value, the power manager sends a device set-power IRP, specifying the device power state that the driver passed in its call to **PoRegisterDeviceForIdleDetection**. The power manager does not send a query IRP before sending the set-power IRP. The drivers in the stack handle the set-power IRP as they would handle any other; they must complete it in a timely manner and they cannot fail it. (See Handling Device Power-Down IRPs.)

When the driver no longer requires idle detection or is not prepared to handle device power-down IRPs, it should call **PoRegisterDeviceForIdleDetection** again, passing zero for both time-out values. Setting the time-outs to zero disables idle detection for both conservation (battery) and performance (AC) power policies. The driver can quickly reenable idle detection; it simply calls **PoRegisterDeviceForIdleDetection** with nonzero time-out values. Once the driver has registered the device, it can enable and disable idle detection by changing the time-out values, even if the device has been powered down and reawakened.

Performing Device-Specific Idle Detection

6/25/2019 • 2 minutes to read • Edit Online

Instead of using the power manager's idle detection routines, a driver can perform its own idle detection based on device-specific criteria.

Such a driver should put its idle device in the lowest possible sleep state that is valid for the current system power state. To do so, the driver requests a power IRP (**PoRequestPowerIrp**) with minor IRP code **IRP_MN_SET_POWER**, specifying the device power state to which the device should transition.

Supporting D3cold in a Driver

6/25/2019 • 3 minutes to read • Edit Online

Starting with Windows 8, the D3 (off) device power state is divided into two distinct substates, D3hot and D3cold. D3 is the lowest-powered device power state, and D3cold is the lowest-powered substate of D3. Moving idle devices to the D3cold substate can reduce power consumption and extend the time that a mobile hardware platform can run on a battery charge.

In D3hot, the device is mostly turned off. However, the device is not disconnected from its main power source, and the parent bus controller can detect the presence of the device on the bus. In D3cold, the main power source is removed from the device, and the bus controller cannot detect the presence of the device. For more information, see the descriptions of D3hot and D3cold in Device Low-Power States.

In earlier versions of Windows, the D3 device power state is implicitly divided into D3hot and D3cold substates, but a device cannot enter D3cold unless the computer is preparing to exit the S0 system power state and enter one of the sleeping states, S1 through S4. The low-power Dx states that a device can enter when the computer is to remain in S0 are limited to D1 through D3hot.

Windows 8 is the first version of Windows to support device-power-state transitions to the D3cold substate when the computer is in S0 and is not preparing to enter a sleeping state. A device that supports D3cold in this way helps to save power in the following ways:

- The device consumes less power in D3cold than in any other low-power Dx state.
- If this device shares a bus with other devices, and all these devices support D3cold, then after all the devices on the bus enter D3cold, the bus controller can enter a low-power Dx state.
- If this device shares a power source with other devices, and all these devices support D3cold, then when the last of these devices enters D3hot, the power source can be removed, at which time these devices all enter D3cold in unison.

Conversely, a device that cannot idle in D3cold can prevent other devices from entering D3cold or other low-power Dx states.

The following topics contain more information about supporting D3cold in a device driver.

In this section

TOPIC	DESCRIPTION
Enabling Transitions to D3cold	All versions of Windows enable a device to be in D3cold while the computer is sleeping (in one of the system low-power states, S1 through S4). Before the computer exits S0, the function drivers, bus drivers, and filter drivers work together to move the device to D3hot. When the computer enters the low-power Sx state, this transition has the side effect of moving the device from D3hot to D3cold.

TOPIC	DESCRIPTION
D3cold Capabilities of a Device	Before the driver that is the power policy owner (PPO) for a device enables the device to enter D3cold (when the computer is to remain in S0), the driver must verify that the device will be responsive and continue to operate correctly after the device enters D3cold.
Using the GUID_D3COLD_SUPPORT_INTERFACE Driver Interface	Starting with Windows 8, drivers can call the routines in the GUID_D3COLD_SUPPORT_INTERFACE interface to determine the D3cold capabilities of devices and to enable these devices to use D3cold. The two primary routines in this interface are *SetD3ColdSupport* and *GetIdleWakeInfo*.
Surprise Wake-Up	A surprise wake-up is an unexpected transition to D0. After a device enters D3cold, it might experience a surprise wake-up as a side effect when the driver for another device on the same power rail requests a transition from D3cold to D0. The driver for the first device must receive notification of the surprise wake-up to prevent the device from remaining in an uninitialized D0 state.

Enabling Transitions to D3cold

6/25/2019 • 3 minutes to read • Edit Online

All versions of Windows enable a device to be in D3cold while the computer is sleeping (in one of the system low-power states, S1 through S4). Before the computer exits S0, the function drivers, bus drivers, and filter drivers work together to move the device to D3hot. When the computer enters the low-power Sx state, this transition has the side effect of moving the device from D3hot to D3cold.

Starting with Windows 8, a device can enter and exit D3cold while the computer remains in S0. The driver that is the power policy owner (PPO) for a device can enable and disable these transitions to D3cold. A driver should not enable its device to enter D3cold unless the device can, if required, wake from D3cold, and then resume normal operation after the transition to D0.

When a device enters D3, it initially enters the D3hot substate of D3. From D3hot, the device can enter either D0 or D3cold. In response to a wake event or I/O request, the device enters D0 from D3hot. Otherwise, the device might remain in D3hot, or it might move from D3hot to D3cold. For more information about these transitions, see the device power state diagram in Device Power States.

The driver does not initiate the device's transition from D3hot to D3cold. Instead, this transition occurs when all the other devices that share a common power source with this device are in D3hot and are prepared to enter D3cold. When the last of these devices enters D3hot, the underlying bus drivers and system firmware remove the power source and the devices enter D3cold in unison.

The PPO driver for a device tells the operating system whether to enable the device's transition from D3hot to D3cold. The driver can supply this information in the INF file that installs the device, or the driver can call the *SetD3ColdSupport* routine at run time to dynamically enable or disable the device's transitions to D3cold. For more information, see Using the GUID_D3COLD_SUPPORT_INTERFACE Driver Interface.

By enabling a device to enter D3cold, a driver guarantees the following behavior:

- The device can tolerate a transition from D3hot to D3cold when the computer is to remain in S0.
- The device will work properly when it returns to D0 from D3cold.

A device that fails to meet either requirement might, after entering D3cold, be unavailable until the computer is restarted or enters a sleeping state. If the device must be able to signal a wake event from any low-power Dx state that it enters, entry to D3cold must not be enabled unless the driver is certain that the device's wake signal will work in D3cold.

Putting a device in D3cold doesn't necessarily mean that all sources of power to the device have been removed; it means only that the sources of power that allow communication to the device through the bus are gone. The device might still be able to draw enough power to signal a wake event to the processor. For example, an Ethernet network interface card (NIC) whose main power source is removed might draw power from the Ethernet cable.

Because D3cold is a state where the bus cannot be used to communicate with the device, a driver can't put its device into D3cold directly. Instead, the driver first calls the **PoRequestPowerIrp** routine to request a D3 power IRP (an **IRP_MN_SET_POWER** request with target state = **PowerDeviceD3**) to move the device from D0 to D3hot. After entering D3hot, the device may or may not move from D3hot to D3cold. The device enters D3cold only when power to the bus is removed, which occurs if the parent bus driver turns off the bus or if the system firmware turns off power to a section of the hardware platform.

D3cold Capabilities of a Device

6/25/2019 • 2 minutes to read • Edit Online

Before the driver that is the power policy owner (PPO) for a device enables the device to enter D3cold (when the computer is to remain in S0), the driver must verify that the device will be responsive and continue to operate correctly after the device enters D3cold.

For a Plug and Play (PnP) device, the operating system typically gets information about the D3cold capabilities of the device from the parent bus driver.

For example, if a device is attached to a PCI or PCI Express bus, the device's PCI configuration space contains a Power Management Register Block that indicates the capabilities of the device. Capability flags in this block specify the device power states from which the device can signal a power management event, or PME (the PCI term for a wake event). These states might include D3hot and D3cold. For more information about PCI power management, see the PCI Bus Power Management Interface Specification.

If a device must be able to signal a wake event from any low-power Dx state that it enters, the device should not enter D3cold unless the device, parent bus controller, and hardware platform support signaling a wake event from D3cold.

The KMDF driver for a device calls the **WdfDeviceAssignS0IdleSettings** method to enable the device to idle in the lowest-powered device power state from which the device can signal a wake event. Starting with KMDF version 1.11, **WdfDeviceAssignS0IdleSettings** includes D3cold in the range of possible low-power Dx states. This method enables a device to idle in D3cold only if the device, the parent bus driver, and the ACPI system firmware support signaling wake events from D3cold.

The WDM driver for a device must decide which low-power Dx state to move the device to when the device is idle. (In contrast, **WdfDeviceAssignS0IdleSettings** automatically selects this Dx state so that the driver does not have to.) If the device must be able to signal a wake event from any low-power Dx state that it enters, the driver can call the *GetIdleWakeInfo* routine to determine the lowest-powered device power state from which the device can signal a wake event. To get this information, *GetIdleWakeInfo* queries the underlying bus driver and ACPI system firmware. Based on the information from *GetIdleWakeInfo*, the driver can call the *SetD3ColdSupport* routine to enable or disable the device's transitions to D3cold.

A device might not require the ability to signal a wake event from D3cold. The device might instead be required to make the transition from D3cold to D0 only in response to software-initiated actions. For example, the driver might need to wake the device if the driver receives an I/O request for the device. With few exceptions, the driver for such a device can enable the device to enter D3cold. A possible exception is a device that requires a large amount of time to make a transition from D3cold to D0. For example, a display device might contain a large amount of memory that needs to be saved before the device enters D3cold and restored after the device exits D3cold.

For more information about ACPI support for D3cold, see Firmware Requirements for D3cold.

Using the GUID_D3COLD_SUPPORT_INTERFACE Driver Interface

6/25/2019 • 5 minutes to read • Edit Online

Starting with Windows 8, drivers can call the routines in the GUID_D3COLD_SUPPORT_INTERFACE interface to determine the D3cold capabilities of devices and to enable these devices to use D3cold. The two primary routines in this interface are *SetD3ColdSupport* and *GetIdleWakeInfo*.

The GUID_D3COLD_SUPPORT_INTERFACE driver interface provides support for the D3cold substate of the D3 device power state. D3 is divided into two substates, D3hot and D3cold. D3 is the lowest-powered device power state, and D3cold uses less power than D3hot. A device can enter D3cold only if the device, the parent bus driver, and the platform firmware support this state. A device that supports D3cold can enter and exit this state when the computer is in the S0 (working) system power state.

The driver that is the power policy owner (PPO) for the device calls the routines in this interface to do the following:

- Discover whether the device, the parent bus driver, and platform firmware support transitions to the D3cold substate.
- Discover whether the device can signal a wake event to the processor when the device is in the D3cold substate.
- Enable and disable transitions to the D3cold substate by the device.

To query for this interface, a device driver sends an IRP_MN_QUERY_INTERFACE IRP down the driver stack. For this IRP, the driver sets the InterfaceType input parameter to GUID_D3COLD_SUPPORT_INTERFACE. On successful completion of the IRP, the Interface output parameter is a pointer to a D3COLD_SUPPORT_INTERFACE structure. This structure contains pointers to the routines in the interface.

For more information about the D3cold device power state, see Supporting D3cold in a Driver.

A driver calls the *SetD3ColdSupport* routine to dynamically enable and disable a device's transitions to D3cold that can occur when the computer is in S0. If the device must be able to signal a wake event from any low-power Dx state that the device enters, the driver should enable the device to enter D3cold only if the device can signal wake events from D3cold. Otherwise, after the device enters D3cold, it might be unavailable until the computer leaves the S0 state.

By default, before the first call to the *SetD3ColdSupport* routine, D3hot-to-D3cold transitions are disabled. To change this default so that D3hot-to-D3cold transitions are enabled before the first *SetD3ColdSupport* call, the driver package for the device can include the following two lines in the DDInstall.HW section of the INF file that installs the driver:

```
Include = machine.inf
Needs = PciD3ColdSupported
```

The *GetIdleWakeInfo* routine enables the driver for a device to discover the device power states from which the device can signal a wake event when the computer is in a particular system power state. The caller to this routine specifies a system power state as an input parameter, and, as an output parameter, the routine reports the lowest-powered device power state from which the device can signal a wait event when the computer is in the specified system power state. For example, the *GetIdleWakeInfo* routine can tell the driver whether the device can signal a wake event from D3cold when the computer is in S0.

The *GetIdleWakeInfo* routine supplies more complete device-wake information than is available from the **IRP_MN_QUERY_CAPABILITIES** request. This request, which all versions of Windows support, supplies a **DEVICE_CAPABILITIES** structure that describes the capabilities of a device. The **DeviceWake** member of this structure contains a subset of the information that is available from the *GetIdleWakeInfo* routine. This member indicates the lowest-powered device power state from which a device can signal a wait event. The information in this member is guaranteed to be accurate only if the computer is in the system low-power state that is indicated by the **SystemWake** member of the structure. If **SystemWake** = **PowerSystemSleeping3**, the information in **DeviceWake** is known to be valid for S3, might frequently be valid for S1 and S2, and might even be valid for S0.

However, as a best practice, a driver should not assume that the information in the **DeviceWake** method is valid for any system power state other than the state indicated by **SystemWake**. For some devices, the lowest Dx state from which a device can signal a wake event varies according to whether the computer is in working state S0 or in a low-power state (S1, S2, S3, or S4). For other devices, the buses to which the devices are connected can handle wake signals when the computer is in S0, but the devices cannot. Only the *GetIdleWakeInfo* routine can accurately describe the device-wake capabilities of these devices.

For example, the PCI Express Base 3.0 Specification defines two separate mechanisms to signal wake events—one mechanism is used when the PCI Express link (bus) is turned on, and the other is used when the link is turned off. When the link is turned on, the device sends a stream of PM_PME Transaction Layer Packets (TLPs) to signal that the device should move from a low-power Dx state to D0. When the link is turned off, the device requests that the link be turned on so that the device can send PM_PME TLPs. To request that the link be turned on, the device either asserts its WAKE# signal (for the more common device form factor) or uses the "beaconing" mechanism (less common).

The PCI Express specification requires that all devices that advertise the ability to signal power management events (PMEs) from D3cold implement both of these device-wake mechanisms, but a driver developer might need to enable a device that does not correctly implement these mechanisms.

If the device can correctly deliver PM_PME TLPs when the link is turned on, the driver can enable the device to enter D3hot when the computer is in S0. If the device can correctly assert its WAKE# signal to turn the link on and then use PM_PME TLPs to initiate the transition to D0, the driver can enable the device to enter D3cold when the computer is in S0.

However, the driver should not enable the device to enter either D3hot or D3cold if the system firmware (the BIOS) can't guarantee that the PCI Express device-wake mechanisms are correctly handled by the hardware platform. A driver can call the *GetIdleWakeInfo* routine to discover whether the firmware claims support for these mechanisms. If a driver uses Kernel-Mode Driver Framework (KMDF) 1.11 or later, a convenient alternative to calling *GetIdleWakeInfo* is to allow the **WdfDeviceAssignS0IdleSettings** method to enable the device to idle in the lowest-powered Dx state from which the device can signal a wake event.

Surprise Wake-Up

6/25/2019 • 4 minutes to read • Edit Online

A surprise wake-up is an unexpected transition to D0. After a device enters D3cold, it might experience a surprise wake-up as a side effect when the driver for another device on the same power rail requests a transition from D3cold to D0. The driver for the first device must receive notification of the surprise wake-up to prevent the device from remaining in an uninitialized D0 state.

When a device moves from D3hot to D3cold, it probably does so because the power source that it shares with some number of other devices was turned off. Some time after these devices enter D3cold, the driver for one of the devices might request a transition to D0. In response to this request, the parent bus driver or ACPI filter driver turns on the power source, and all the devices that share the power source enter their default, power-on hardware states.

The only device driver that expects this power state change is the driver that requested the change. The drivers for the other devices must receive notification of this change so that they can properly initialize their devices to operate in D0. Only a driver that can receive this notification should enable its device to enter D3cold. Otherwise, the driver will not know when the device enters D0.

When a device is turned on, it enters a default, uninitialized hardware state. For example, the PCI Express Base 3.0 Specification defines a *D0 uninitialized* state that a device enters when it first receives power. The definition of this state is specific to PCI and PCI Express devices, but devices that connect to other buses are designed to enter similar hardware states when they are turned on.

In the case of a PCI or PCI Express device that implements multiple functions, these device functions probably share the same power rail. However, each function might have a separate driver and the drivers for these functions are unlikely to communicate directly with each other. When the driver for one of these functions requests a power state change from D3cold to D0, the drivers for the other functions do not expect this change. When these other functions receive power, their drivers must be notified so that they can configure the functions to operate correctly in D0.

A bus driver detects when power to a child device is being turned on. If this device's function driver did not request a transition to D0, the bus driver prompts the device driver to send itself a D0 power IRP (an **IRP_MN_SET_POWER** request with target state = **PowerDeviceD0**) to initialize the device to operate in D0. From this initialized D0 state, the device driver can then initiate the device's transition to D3hot. Device drivers can receive notifications of surprise transitions to D0 from bus drivers in the following ways:

- Device drivers that directly or indirectly register themselves as clients of the run-time power management framework (PoFx) receive notification callbacks.
- Drivers for devices that arm their devices for wake have their pending **IRP_MN_WAIT_WAKE** requests completed by the bus drivers.

Starting with Windows 8, a device's function driver, acting as the power policy owner, can register itself as a client of PoFx. When the bus driver notifies PoFx that the device experienced a surprise transition to D0, PoFx helps the device to move to an initialized D0 state, and then to D3hot. First, PoFx calls the driver's *DevicePowerRequiredCallback* routine to prompt the device driver to send a D0 power IRP down the device stack. Next, PoFx calls the driver's *DevicePowerNotRequiredCallback* routine to notify the device driver that the device is not required to stay in the D0 state.

Starting with Kernel-Mode Driver Framework (KMDF) version 1.11, the KMDF driver for a single-component device can indirectly register itself with PoFx by calling the **WdfDeviceWdmAssignPowerFrameworkSettings** method. In this call, the driver supplies pointers to callback routines that notify the driver of surprise transitions to

D0. For more information, see Supporting Functional Power States.

A driver that does not register its device with PoFx can still be notified of a surprise transition to D0 if the device is armed for wake. When the bus drivers turn on the power to the device, they complete the driver's **IRP_MN_WAIT_WAKE** request. In response, the driver initializes its device to operate in D0. The device is likely to be idle, in which case the driver, after some time, will move this device to D3hot.

A function driver that does not register with PoFx and that does not arm its device for wake receives no notification of a surprise transition from D3cold to D0. The device might spend large amounts of time in an uninitialized D0 state. In this state, all of the components in the device are typically turned on. To reduce power consumption by idle devices, drivers should enable entry to D3cold only if they can receive notifications of surprise transitions to D0.

Handling System Power State Requests

12/5/2018 • 2 minutes to read • Edit Online

All drivers must be able to respond to system power state requests if the system is to sleep, hibernate, and wake successfully. A driver for a device changes the device power state for the device in response to system power state requests.

If any driver does not support system power management, individual devices can sleep and wake, but the power manager cannot put the system as a whole into a sleeping state.

The following topics cover details of handling system power state requests:

System Power States

System Power Policy

Preventing System Power State Changes

Handling IRP_MN_QUERY_POWER for System Power States

Handling IRP_MN_SET_POWER for System Power States

System Power States

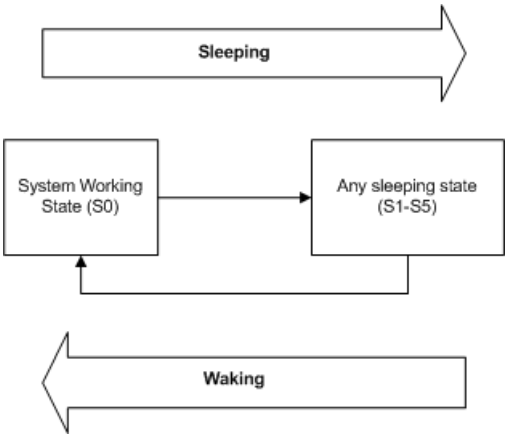
12/5/2018 • 2 minutes to read • Edit Online

System power states describe the power consumption of the system as a whole. The operating system supports six system power states, referred to as S0 (fully on and operational) through S5 (power off). Each state is characterized by the following:

- Power consumption: how much power does the computer use?
- Software resumption: from what point does the operating system restart?
- Hardware latency: how long does it take to return the computer to the working state?
- System hardware context (such as the content of volatile processor registers, memory caches, and RAM): how much system hardware context is retained? Must the operating system reboot to return to the working state?

State S0 is the working state. States S1, S2, S3, and S4 are sleeping states, in which the computer appears off because of reduced power consumption but retains enough context to return to the working state without restarting the operating system. State S5 is the shutdown or off state.

A system is *waking* when it is in transition from the shutdown state (S5) or any sleeping state (S1-S4) to the working state (S0), and it is going to sleep when it is in transition from the working state to any sleep state or the shutdown state. The following figure shows the possible system power state transitions.



As the previous figure shows, the system cannot enter one sleep state directly from another; it must always enter the working state before entering any sleep state. For example, a system cannot transition from state S2 to S4, nor from state S4 to S2. It must first return to S0, from which it can enter the next sleep state. Because a system in an intermediate sleep state has already lost some operating context, it must return to the working state to restore that context before it can make an additional state transition.

System Working State S0

12/5/2018 • 2 minutes to read • Edit Online

System power state S0, the system working state, has the following characteristics:

Power consumption
Maximum. However, the power state of individual devices can change dynamically as power conservation takes place on a per-device basis. Unused devices can be powered down and powered up as needed.

Software resumption
Not applicable.

Hardware latency
None.

System hardware context
All context is retained.

System Sleeping States

10/7/2019 • 3 minutes to read • Edit Online

States S1, S2, S3, and S4 are the sleeping states. A system in one of these states is not performing any computational tasks and appears to be off. Unlike a system in the shutdown state (S5), however, a sleeping system retains memory state, either in the hardware or on disk. The operating system need not be rebooted to return the computer to the working state.

Some devices can wake the system from a sleeping state when certain events occur, such as an incoming call to a modem. In addition, on some computers, an external indicator tells the user that the system is merely sleeping.

With each successive sleep state, from S1 to S4, more of the computer is shut down. All ACPI-compliant computers shut off their processor clocks at S1 and lose system hardware context at S4 (unless a hibernate file is written before shutdown), as listed in the sections below. Details of the intermediate sleep states can vary depending on how the manufacturer has designed the machine. For example, on some machines certain chips on the motherboard might lose power at S3, while on others such chips retain power until S4. Furthermore, some devices might be able to wake the system only from S1 and not from deeper sleep states.

System Power State S1

System power state S1 is a sleeping state with the following characteristics:

Power consumption
Less consumption than in S0 and greater than in the other sleep states. Processor clock is off and bus clocks are stopped.

Software resumption
Control restarts where it left off.

Hardware latency
Typically no more than two seconds.

System hardware context
All context retained and maintained by hardware.

System Power State S2

System power state S2 is similar to S1 except that the CPU context and contents of the system cache are lost because the processor loses power. State S2 has the following characteristics:

Power consumption
Less consumption than in state S1 and greater than in S3. Processor is off. Bus clocks are stopped; some buses might lose power.

Software resumption
After wake-up, control starts from the processor's reset vector.

Hardware latency
Two seconds or more; greater than or equal to the latency for S1.

System hardware context
CPU context and system cache contents are lost.

System Power State S3

System power state S3 is a sleeping state with the following characteristics:

Power consumption
Less consumption than in state S2. Processor is off and some chips on the motherboard also might be off.

Software resumption
After the wake-up event, control starts from the processor's reset vector.

Hardware latency
Almost indistinguishable from S2.

System hardware context
Only system memory is retained. CPU context, cache contents, and chipset context are lost.

System Power State S4

System power state S4, the hibernate state, is the lowest-powered sleeping state and has the longest wake-up latency. To reduce power consumption to a minimum, the hardware powers off all devices. Operating system context, however, is maintained in a hibernate file (an image of memory) that the system writes to disk before entering the S4 state. Upon restart, the loader reads this file and jumps to the system's previous, prehibernation location.

If a computer in state S1, S2, or S3 loses all AC or battery power, it loses system hardware context and therefore must reboot to return to S0. A computer in state S4, however, can restart from its previous location even after it loses battery or AC power because operating system context is retained in the hibernate file. A computer in the hibernate state uses no power (with the possible exception of trickle current).

State S4 has the following characteristics:

Power consumption
Off, except for trickle current to the power button and similar devices.

Software resumption
System restarts from the saved hibernate file. If the hibernate file cannot be loaded, rebooting is required. Reconfiguring the hardware while the system is in the S4 state might result in changes that prevent the hibernate file from loading correctly.

Hardware latency
Long and undefined. Only physical interaction returns the system to the working state. Such interaction might include the user pressing the ON switch or, if the appropriate hardware is present and wake-up is enabled, an incoming ring for the modem or activity on a LAN. The machine can also awaken from a resume timer if the hardware supports it.

System hardware context
None retained in hardware. The system writes an image of memory in the hibernate file before powering down. When the operating system is loaded, it reads this file and jumps to its previous location.

System Shutdown State S5

12/5/2018 • 2 minutes to read • Edit Online

In the S5, or shutdown, state, the machine has no memory state and is not performing any computational tasks.

The only difference between states S4 and S5 is that the computer can restart from the hibernate file in state S4, while restarting from state S5 requires rebooting the system.

State S5 has the following characteristics:

Power consumption
Off, except for trickle current to devices such as the power button.

Software resumption
Boot is required upon awakening.

Hardware latency
Long and undefined. Only physical interaction, such as the user pressing the ON switch, returns the system to the working state. The BIOS can also awaken from a resume timer if the system is so configured.

System hardware context
None retained.

System Power Actions

12/5/2018 • 2 minutes to read • Edit Online

When the power manager sends an IRP to set or query the system power state, it specifies a system power state along with an additional parameter that gives information about the power state change. This parameter, passed at **Irp->Parameters.Power.ShutdownType**, is an enumerator of the POWER_ACTION type. The enumerator characterizes the system power state request, as shown in the following table.

POWER_ACTION ENUMERATOR	SYSTEM POWER STATE REQUESTED
PowerActionNone	S0 or no system power IRP active
PowerActionSleep	S1, S2, or S3
PowerActionHibernate	S4
PowerActionShutdown (Microsoft Windows 2000 and later systems only)	S5
PowerActionShutdownReset	S5
PowerActionShutdownOff	S5

When a driver receives a system query or set-power IRP for S5, it can check **ShutdownType** For more information about the requested shutdown. A driver can use this information to optimize its shutdown sequence when the machine is resetting instead of shutting off power indefinitely. Drivers of most devices retain power when the system resets. However, for certain devices, such as a video streaming device that performs direct memory access (DMA), a driver might choose to power down its device when the system is resetting, thus stopping any ongoing I/O.

When a device power policy owner sends a *device* power IRP to its device stack in response to a system power IRP, drivers can use the **ShutdownType** parameter to get information about the current *system* power IRP. In this case, the value of **ShutdownType** indicates the currently requested system power state, or it is **PowerActionNone** if a system request is not outstanding. Drivers should not, however, rely on this information if the device IRP requests state D0.

In Windows 98/Me, this member always contains **PowerActionNone** when the IRP requests a device power state.

System Power Policy

6/25/2019 • 2 minutes to read • Edit Online

In its role as system power policy manager, the power manager keeps track of system activity, determines the appropriate system power state, and sends **IRP_MJ_POWER** requests to query or change the system power state. It also provides interfaces through which applications can read and write power policy settings (see the Microsoft Windows SDK).

The power manager maintains two separate power policies — one for AC (wall current) and one for DC (battery or UPS) — and automatically switches between these two policies depending on the current power source. Typically, AC power policy emphasizes performance over conservation, while DC power policy emphasizes conservation over performance. To find out when the system changes from one policy to the other, a driver can register for notification with the system's \Callback\PowerState callback object. For further information, see **ExCreateCallback** and Callback Objects.

Computers that comply with the APCI specification automatically switch from AC to battery power, and from one battery to another, as each such power source goes off line. If the computer hardware allows the operating system to select the power source, the power manager tracks which battery is the least charged but still functional and selects it to power the computer.

As soon as AC power becomes available, the computer hardware automatically starts to charge a battery. If the hardware allows the operating system to select which battery to charge, the power manager selects the least discharged battery for recharging; this increases the chances that the system will have at least one well-charged battery at all times.

Regardless of any other settings, the power manager carries out the DC power policy for a critical battery if a battery that is rechargeable or supplies system power reports the hardware condition "critical" and is in the discharging state for two seconds or longer. Power policy in this situation typically requires a transition to the hibernate or shutdown state.

Preventing System Power State Changes

6/25/2019 • 2 minutes to read • Edit Online

Although drivers cannot directly set system power policy, the power manager provides three routines through which a driver can prevent system transitions out of the working state: **PoSetSystemState**, **PoRegisterSystemState**, and **PoUnregisterSystemState**.

By calling **PoRegisterSystemState** or **PoSetSystemState**, a driver can notify the power manager that a user is present or that the driver requires use of the system or display.

PoRegisterSystemState allows a driver to register a continuous busy state. It returns a handle through which the driver can later change its settings. As long as the state registration is in effect, the power manager does not attempt to put the system to sleep. The driver cancels the state registration by calling **PoUnregisterSystemState**.

With **PoSetSystemState**, a driver notifies the power manager of the same conditions (user present, system required, display required), but this setting is not continuous. It has the effect of restarting any idle count downs associated with the specified conditions.

Using these routines, a driver can forestall many, but not all, transitions out of the working state. The power manager always shuts down the system when loss of power is imminent or when a user explicitly requests shutdown.

Handling IRP_MN_QUERY_POWER for System Power States

6/25/2019 • 2 minutes to read • Edit Online

The power manager sends a power IRP with the minor IRP code **IRP_MN_QUERY_POWER** and **SystemPowerState** in **Parameters.Power.Type** to determine whether it can safely change to a specified system power state (S1-S5) and to allow drivers to prepare for such a change.

Whenever possible, the power manager queries before sending an **IRP_MN_SET_POWER** that requests a lower (less powered) state. However, in cases of a failing battery or imminent loss of power, the power manager sends the set-power IRP without querying first. The power manager never sends a query before sending an IRP to set the system in the working state (S0).

For information about how a power policy owner for a device handles system query-power requests, see Handling a System Query-Power IRP in a Device Power Policy Owner.

For information about how drivers (that are not the power policy owner for a device) handle system query-power requests, see the following:

Handling a System Query-Power IRP in a Filter or Function Driver

Failing a System Query-Power IRP in a Filter or Function Driver

Handling a System Query-Power IRP in a Bus Driver

Note that a driver must never send a device **IRP_MN_SET_POWER** request in response to a system query; it requests such an IRP only after it receives a system set-power request.

Because the power manager sends the system query IRP to each device stack on the system, it is possible that a driver for one device might fail the query while drivers for other devices complete it successfully. Beginning with Windows Vista, a change to the system power state to a sleep state is a critical power state change. Even if a driver fails a system query-power IRP, the power manager in Windows Vista might still change the system power state to a sleep state. It is also possible that a battery might expire while a query is active, requiring an immediate shutdown. Consequently, after a query IRP, drivers must be prepared to receive any of the following power IRPs:

- An **IRP_MN_SET_POWER** to the queried state
- An **IRP_MN_SET_POWER** to a different power state
- An **IRP_MN_SET_POWER** to the current power state
- An **IRP_MN_QUERY_POWER** to any state

Usually, however, a driver receives a system set-power IRP following a system query IRP. Regardless, a driver must be ready to change the system power state even if the driver fails a query-power IRP.

Handling a System Query-Power IRP in a Device Power Policy Owner

6/25/2019 • 2 minutes to read • Edit Online

When a device power policy owner receives an **IRP_MN_QUERY_POWER** for a system power state, it responds by passing down the query and, in an *IoCompletion* routine, sending an **IRP_MN_QUERY_POWER** for a device power state. When all drivers in the stack have completed the device query, the device power policy owner completes the system query.

A device power policy owner should take the following steps in its DispatchPower routine to respond to a system query:

1. Call **IoAcquireRemoveLock**, passing the current IRP, to ensure that the driver does not receive a PnP **IRP_MN_REMOVE_DEVICE** request while handling the power IRP.

   If **IoAcquireRemoveLock** returns a failure status, the driver should not continue processing the IRP. Instead, beginning with Windows Vista, the driver should call **IoCompleteRequest** to complete the IRP and return the failure status. In Windows Server 2003, Windows XP, and Windows 2000, the driver should call **PoStartNextPowerIrp**, call **IoCompleteRequest** to complete the IRP, and return the failure status.

2. Ensure that the driver can support the queried system power state, as described in Failing a System Query-Power IRP in a Filter or Function Driver. If not, complete the IRP with a failure status as described in that section.

   However, a driver must not fail a query for S4 (**PowerSystemHibernate**) if its device is enabled for wake-up but it cannot wake the system from the hibernate state. In this case, the power policy owner for the driver (which sent the **IRP_MN_WAIT_WAKE**) must cancel the wait/wake IRP and succeed the system query. For more information, see Canceling a Wait/Wake IRP.

3. If the driver can support the queried system power state, call **IoMarkIrpPending**.

4. Set up the IRP stack location for the next-lower driver by calling **IoCopyCurrentIrpStackLocationToNext**.

5. Set an *IoCompletion* routine in the system query power IRP.

6. Call **IoCallDriver** (in Windows 7 and Windows Vista) or **PoCallDriver** (in Windows Server 2003, Windows XP, and Windows 2000), to pass the IRP to the next-lower driver.

7. Return STATUS_PENDING.

The *IoCompletion* routine should do the following:

1. Check **Irp->IoStatus.Status** to ensure that lower drivers have completed the IRP successfully. If a lower driver has specified a non-success NTSTATUS value, the *IoCompletion* routine should return the NTSTATUS value.

2. If lower drivers have successfully completed the IRP, call **PoRequestPowerIrp** to send a device query-power IRP for a device power state that is valid for the queried system power state. If necessary, consult the DEVICE_STATE array in the **DEVICE_CAPABILITIES** structure to determine which device power states are valid for the queried system power state.

3. Specify a callback routine (*CompletionFunction* parameter) in the call to **PoRequestPowerIrp** and pass the system IRP in the *Context* area.

4. Return STATUS_MORE_PROCESSING_REQUIRED so that the driver can finish processing the system query IRP in the callback routine.

After the IRP has been completed and all *IoCompletion* routines set during IRP processing have been run, the power manager, through the I/O manager, calls the power policy manager's callback routine (the *CompletionFunction* parameter to **PoRequestPowerIrp**). The callback routine, in turn, must do the following:

1. Call **PoStartNextPowerIrp** to start the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.)
2. Complete the system query-power IRP (call **IoCompleteRequest**) with the status returned for the device query-power IRP.
3. Call **IoReleaseRemoveLock** to free the previously acquired lock.

Remember that the device power policy owner not only sends the device query but also must handle it on its way down the device stack. For more information, see Handling IRP_MN_QUERY_POWER for Device Power States.

Handling a System Query-Power IRP in a Filter or Function Driver

6/25/2019 • 2 minutes to read • Edit Online

A filter or function driver (that is not the power policy owner for a device) should pass a system query-power IRP to the next-lower driver, in the following steps:

1. Call **IoAcquireRemoveLock**, passing the current IRP, to ensure that the driver does not receive a PnP **IRP_MN_REMOVE_DEVICE** request while handling the power IRP.

   If **IoAcquireRemoveLock** returns a failure status, the driver should not continue processing the IRP. Instead, beginning with Windows Vista, the driver should call **IoCompleteRequest** to complete the IRP and return the failure status. In Windows Server 2003, Windows XP, and Windows 2000, the driver should call **PoStartNextPowerIrp**, call **IoCompleteRequest** to complete the IRP, and return the failure status.

2. Determine whether it should fail the query. For guidelines, see Failing a System Query-Power IRP in a Filter or Function Driver and complete processing as described in that section.

3. Call **PoStartNextPowerIrp**. (Windows Server 2003, Windows XP, and Windows 2000 only)

4. Set the IRP stack location (**IoSkipCurrentIrpStackLocation** or **IoCopyCurrentIrpStackLocationToNext**). The driver can set an *IoCompletion* routine in the IRP, but doing so is rarely necessary.

5. Call **IoCallDriver** (in Windows 7 and Windows Vista) or **PoCallDriver** (in Windows Server 2003, Windows XP, and Windows 2000) to pass the IRP to the next-lower driver.

6. Call **IoReleaseRemoveLock**. However, if the driver set an *IoCompletion* routine for the IRP, make this call from the *IoCompletion* routine instead.

7. Return STATUS_PENDING from its *DispatchPower* routine.

Failing a System Query-Power IRP in a Filter or Function Driver

6/25/2019 • 2 minutes to read • Edit Online

A filter or function driver (that is not the power policy owner for a device) can fail an **IRP_MN_QUERY_POWER** request if either of the following is true:

- The device is enabled for wake-up and the requested system power state is less powered than the value of **SystemWake**, which specifies the least-powered state from which the device can wake the system. For example, a device that can wake the system from S2 but not from S3 would fail a query for S3 but succeed a query for S2.
- Entering a device power state that corresponds to the requested state would force the driver to abandon an operation that would lose data, such as an open modem connection. A driver rarely will fail a query for this reason; under most circumstances, the application handles such cases.

To fail an **IRP_MN_QUERY_POWER** request for a system power state, a driver should take the following steps:

1. Call **PoStartNextPowerIrp** to indicate that the driver is prepared to handle the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only)
2. Set **Irp->IoStatus.Status** to a failure status and call **IoCompleteRequest**, specifying IO_NO_INCREMENT. Do not pass the IRP further down the device stack.
3. Call **IoReleaseRemoveLock** to release the previously acquired lock.
4. Return a failure status from its *DispatchPower* routine.

Handling a System Query-Power IRP in a Bus Driver

6/25/2019 • 2 minutes to read • Edit Online

When a system query-power request reaches a bus driver (that is not the power policy owner for a device), the driver ensures that it can support a device power state that corresponds to the queried system power state and, if wake-up is enabled, that the queried system power state would not prevent its device from waking the system.

In Windows 7 and Windows Vista, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS if the driver can change to the specified power state or sets a failure status if the driver cannot.

In Windows Server 2003, Windows XP, and Windows 2000, the bus driver first calls **PoStartNextPowerIrp** and then sets **Irp->IoStatus.Status** to STATUS_SUCCESS if the driver can change to the specified power state or sets a failure status if the driver cannot.

After the bus driver completes the IRP, the power manager calls *IoCompletion* routines set by other drivers as they passed the IRP down the stack.

Handling IRP_MN_SET_POWER for System Power States

6/25/2019 • 2 minutes to read • Edit Online

The power manager sends a power IRP that specifies the minor code **IRP_MN_SET_POWER** and a system power state for one of the following reasons:

- To change the system power state.
- To reaffirm the current power state after a failed **IRP_MN_QUERY_POWER** request.

Through the I/O manager, the power manager sends the IRP to the top driver in the device stack at each PnP device node. The IRP notifies all drivers in the stack of the correct system power state.

To ensure an orderly start-up, the power manager sequences system power-up IRPs so that parent devices have the opportunity to power up before their children do. The power manager does not query before sending a system power-up IRP.

Similarly, to ensure that the computer sleeps or shuts down in an orderly way, the power manager sends system IRPs that specify sleep, hibernation, or shutdown in a defined sequence, so that devices farther from the root power down before devices nearer the root. Whenever possible, the power manager queries before sending such an IRP. For more information, see Handling IRP_MN_QUERY_POWER for System Power States.

The system power IRP is not a direct request to change power state — it is a notification. A driver must not change the power state of its device as a direct response to the *system* power IRP; a driver changes its device's power state only in response to a *device* power IRP. (The device power policy owner sends the device power IRP; see Handling a System Set-Power IRP in a Device Power Policy Owner.)

Even if the device is already in a device power state that is valid for the requested system power state, each driver must nevertheless pass the system set-power IRP to the next-lower driver, until it reaches the bus driver. Only the bus driver is allowed to complete this IRP.

How a driver handles this IRP depends upon its role in the device stack, as described in the following sections:

Handling a System Set-Power IRP in a Device Power Policy Owner

Handling a System Set-Power IRP in a Bus Driver

Handling a System Set-Power IRP in a Filter Driver

A driver cannot fail an **IRP_MN_SET_POWER** request to set the system power state. The power manager ignores any failure status returned for this IRP.

Handling a System Set-Power IRP in a Device Power Policy Owner

6/25/2019 • 2 minutes to read • Edit Online

In response to a system set-power IRP, the power policy owner for a device stack is responsible for putting its device stack into an appropriate device power state.

As a general rule, when a device power policy owner receives an **IRP_MN_SET_POWER** for a system power state, it should respond by passing the system set-power IRP down the device stack. A device power policy owner should also respond by sending down the device stack **IRP_MN_SET_POWER** for a corresponding device power state in an *IoCompletion* routine. After all drivers in the stack have completed the device set-power IRP, the device power policy owner completes the system set-power IRP.

However, to improve system resume performance, device power owners for devices that do not have child devices should use a different approach to reduce the time it takes a system to return to working state S0 from a sleeping state. In this case, in response to a system set-power IRP that returns a system to working state S0, device power policy owners should perform the following sequence of operations:

1. After receiving an **IRP_MN_SET_POWER** IRP for the S0 system power state in the driver's DispatchPower routine, set an *IoCompletion* routine for the IRP and pass the IRP down the stack.
2. In the *IoCompletion* routine set in step (1), request an **IRP_MN_SET_POWER** IRP for the corresponding device power state and then immediately complete the system set-power IRP. The driver should not wait for device set-power IRPs to complete before it completes the system set-power IRP. The *IoCompletion* routine is executed after all lower-level drivers have completed the system set-power IRP and the system set-power IRP is passed back to the driver's functional device object (FDO).
3. Perform any required device-specific initialization.
4. Complete the device set-power IRP that was sent in step (2).
5. Process I/O requests that were queued when the device was in a device sleeping state.

The kernel power manager has a limited set of IRP dispatch queues, and must quickly notify all devices in the system of the return to the system working state S0. Drivers that fail to complete the system set-power IRP as quickly as possible prevent other devices from getting their system set-power IRP, which can adversely affect overall system performance during system power-state transitions.

For more detail on handling system set-power IRPs, see the following:

Determining the Correct Device Power State

Sending a Device Set-Power IRP in Response to a System Set-Power IRP

Determining the Correct Device Power State

6/25/2019 • 2 minutes to read • Edit Online

The power policy owner checks the **DeviceState** array in the **DEVICE_CAPABILITIES** structure to determine the valid range of device power states for each system power state. The array lists the highest device power state the underlying device can support for each system power state.

When choosing a specific state from this range, consider the following:

- Most devices enter the D0 state when the system enters the S0 state.
- Most devices enter the D3 state when the system enters any sleeping state. However, a device that is enabled for wake-up might be required to enter D1 or D2 instead, if it supports such states. For further information, see Reporting Device Power Capabilities.
- Special rules apply for the device that will hold the hibernation file. If the system IRP requests **PowerSystemHibernate**, the device that will hold the hibernation file must not power off. The power policy owner for such a device should request device power state D3 and save context, but the device's drivers must not power off the device.

If the system IRP requests **PowerSystemShutdown**, the driver should check the POWER_ACTION value at **Irp->Parameters.Power.ShutdownType** to determine the reason for the state change. For further information, see System Power Actions.

The device power policy owner must send a device set-power IRP for each system set-power IRP, even if the device is already in the correct device power state. If the driver previously suspended device operations in response to a query-power IRP, the set-power IRP notifies it to stop queuing IRPs and return to normal operation for its current power state. The only exception occurs when the device is in the D3 state; in this case, the driver need not send an additional **IRP_MN_SET_POWER** request for D3.

Sending a Device Set-Power IRP in Response to a System Set-Power IRP

6/25/2019 • 2 minutes to read • Edit Online

The device power policy owner should take the following steps to respond to a system set-power IRP:

1. Call **IoAcquireRemoveLock**, passing the current IRP as the *Tag* parameter, to ensure that the driver does not receive a Plug and Play **IRP_MN_REMOVE_DEVICE** request while handling the power IRP.

   If **IoAcquireRemoveLock** returns a failure status, the driver should not continue processing the IRP. Instead, starting with Windows Vista, the driver should call **IoCompleteRequest** to complete the request and then return the failure status. In Windows Server 2003, Windows XP, and Windows 2000, the driver should first call **PoStartNextPowerIrp**, call **IoCompleteRequest** to complete the IRP, and then return the failure status.

2. Set up the IRP stack location for the next-lower driver by calling **IoCopyCurrentIrpStackLocationToNext**.

3. Set an *IoCompletion* routine in the system set-power IRP.

4. Call **IoMarkIrpPending** to mark the system set-power IRP as pending.

5. Call **IoCallDriver** (starting with Windows Vista) or **PoCallDriver** (in Windows Server 2003, Windows XP, and Windows 2000) to pass the system set-power IRP to the next-lower driver.

6. Return STATUS_PENDING from its *DispatchPower* routine.

In the *IoCompletion* routine (see Step 3 in the preceding list), the device power policy owner sends a device set-power IRP as follows:

1. Inspect the system set-power IRP to get the requested system power state. Choose an appropriate device power state for that system power state. For further information, see Determining the Correct Device Power State.

2. Call **PoRequestPowerIrp** to send an **IRP_MN_SET_POWER** for the device power state determined in Step 1. The power policy owner must send the device set-power request even if the device is already in that state.

3. Specify a power-completion callback routine (*CompletionFunction*) in the call to **PoRequestPowerIrp** and pass the system set-power IRP in the *Context* buffer.

4. Return STATUS_MORE_PROCESSING_REQUIRED from the *IoCompletion* routine, so that the driver can finish processing the system set-power IRP in the power-completion callback routine.

Remember that the device power policy owner not only sends the device set-power IRP but also must handle this IRP as it travels through the device stack. Consequently, a device power policy owner might have not only a power-completion callback routine associated with the device set-power IRP and an *IoCompletion* routine for the system set-power IRP, but also an *IoCompletion* routine for the device set-power IRP. For further information, see Handling IRP_MN_SET_POWER for Device Power States.

After the I/O manager calls all the *IoCompletion* routines that were set as the device set-power IRP traveled down the device stack, the I/O manager calls the power-completion callback routine. By this time, all drivers in the stack have completed the device set-power IRP and the device power transition is complete.

The power-completion callback routine must do the following:

1. Call **PoStartNextPowerIrp** to start the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.)
2. Complete the system set-power IRP (**IoCompleteRequest**) with the status returned for the device set-power IRP.
3. Call **IoReleaseRemoveLock** to free the previously acquired lock.
4. Return the status with which the set-power IRPs completed.

Handling a System Set-Power IRP in a Bus Driver

6/25/2019 • 2 minutes to read • Edit Online

When a bus driver receives a system set-power IRP, it must take the following steps:

1. Call **PoStartNextPowerIrp** to start the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.)
2. Set **Irp->IoStatus.Status** to STATUS_SUCCESS. The driver cannot fail a system set-power IRP.
3. Call **IoCompleteRequest**, specifying IO_NO_INCREMENT, to complete the IRP.

The bus driver does not change device power settings until it receives a power IRP that requests a device power state.

Handling a System Set-Power IRP in a Filter Driver

6/25/2019 • 2 minutes to read • Edit Online

All filter drivers and any function driver that does not own power policy for its device stack should simply pass the system set-power IRP to the next-lower driver, in the following steps:

1. Call **IoAcquireRemoveLock**, passing the current IRP, to ensure that the driver does not receive a PnP **IRP_MN_REMOVE_DEVICE** request while handling the power IRP.

   If **IoAcquireRemoveLock** returns a failure status, the driver should not continue processing the IRP. Instead, beginning with Windows Vista, the driver should call **IoCompleteRequest** to complete the IRP and return the failure status. In Windows Server 2003, Windows XP, and Windows 2000, the driver should first call **PoStartNextPowerIrp**, call **IoCompleteRequest** to complete the IRP, and then return the failure status.

2. Call **PoStartNextPowerIrp** to start the next power IRP. (Windows Server 2003, Windows XP, and Windows 2000 only.)

3. Set the IRP stack location (**IoSkipCurrentIrpStackLocation** or **IoCopyCurrentIrpStackLocationToNext**). The driver can set an *IoCompletion* routine in the IRP, but doing so is rarely necessary.

4. Call **IoCallDriver** (in Windows 7 and Windows Vista) or **PoCallDriver** (in Windows Server 2003, Windows XP, and Windows 2000) to pass the IRP to the next-lower driver.

5. Call **IoReleaseRemoveLock**. However, if the driver set an *IoCompletion* routine for the IRP, make this call from the *IoCompletion* routine instead.

6. Return STATUS_PENDING from its *DispatchPower* routine.

Overview of the Power Management Framework

12/20/2018 • 2 minutes to read • Edit Online

Starting with Windows 8, the run-time power management framework (PoFx) supports power and clock management at the component (or subdevice) level. A device driver registers with PoFx to independently manage power usage in the individual components in a device. PoFx provides the fine-grained control necessary to extend the time that a Windows portable computer, tablet, phone, or other mobile device can run on a battery charge. PoFx reduces power usage in a way that maintains the appearance of a mobile device that is always on and always connected.

A component, or subdevice, is a functional hardware unit in a device that can be turned on or switched to a low-power state independently of the other components in the same device. For example, an audio device might have separate components for playback and recording whose power states can be managed independently of each other. If the playback component is being used, but the recording component is idle, the recording component can be switched to a low-power state without interfering with the activity of the playback component.

Device power management reference

5/6/2019 • 7 minutes to read • Edit Online

Drivers can divide their device hardware into multiple logical components to enable fine-grained power management. A component has a set of power states that can be managed independently of the power states of other components in the same device. In the F0 state, the component is fully turned on. The component might support additional, low-power states F1, F2, and so on.

The power policy owner for a device is typically the device's function driver. To enable component-level power management, this driver registers the device with the power management framework (PoFx). By registering the device, the driver assumes the responsibility for informing PoFx when a component is actively being used and when the component is idle. PoFx makes intelligent idle state choices for the device based on information about the component activity, latency tolerance, expected idle durations, and wake requirements. By controlling power usage at the component level, PoFx can reduce power requirements while preserving system responsiveness. For more information, see Component-Level Power Management.

Device Power Management Routines

These routines are implemented by the power management framework (PoFx) to enable device power management. These routines are called by the driver that is the power policy owner (PPO) for a device. Typically, the function driver for a device is the PPO for this device.

TOPIC	DESCRIPTION
PoFxActivateComponent	The **PoFxActivateComponent** routine increments the activation reference count on the specified component.
PoFxCompleteDevicePowerNotRequired	The **PoFxCompleteDevicePowerNotRequired** routine notifies the power management framework (PoFx) that the calling driver has completed its response to a call to the driver's *DevicePowerNotRequiredCallback* callback routine.
PoFxCompleteIdleCondition	The **PoFxCompleteIdleCondition** routine informs the power management framework (PoFx) that the specified component has completed a pending change to the idle condition.
PoFxCompleteIdleState	The PoFxCompleteIdleState routine informs the power management framework (PoFx) that the specified component has completed a pending change to an Fx state.
PoFxIdleComponent	The **PoFxIdleComponent** routine decrements the activation reference count on the specified component.
PoFxIssueComponentPerfStateChange	The **PoFxIssueComponentPerfStateChange** routine submits a request to place a device component in a particular performance state.
PoFxIssueComponentPerfStateChangeMultiple	The **PoFxIssueComponentPerfStateChangeMultiple** routine submits a request to change the performance states in multiple performance state sets simultaneously for a device component.

TOPIC	DESCRIPTION
PoFxNotifySurprisePowerOn	The **PoFxNotifySurprisePowerOn** routine notifies the power management framework (PoFx) that a device was turned on as a side effect of supplying power to some other device.
PoFxPowerControl	The **PoFxPowerControl** routine sends a power control request to the power management framework (PoFx).
PoFxQueryCurrentComponentPerfState	The **PoFxQueryCurrentComponentPerfState** routine retrieves the active performance state in a component's performance state set.
PoFxRegisterComponentPerfStates	The **PoFxRegisterComponentPerfStates** routine registers a device component for performance state management by the power management framework (PoFx).
PoFxRegisterDevice	The **PoFxRegisterDevice** routine registers a device with the power management framework (PoFx).
PoFxReportDevicePoweredOn	The **PoFxReportDevicePoweredOn** routine notifies the power management framework (PoFx) that the device completed the requested transition to the D0 (fully on) power state.
PoFxSetComponentLatency	The **PoFxSetComponentLatency** routine specifies the maximum latency that can be tolerated in the transition from the idle condition to the active condition in the specified component.
PoFxSetComponentResidency	The **PoFxSetComponentResidency** routine sets the estimated time for how long a component is likely to remain idle after the component enters the idle condition.
PoFxSetComponentWake	The **PoFxSetComponentWake** routine indicates whether the driver arms the specified component to wake whenever the component enters the idle condition.
PoFxSetDeviceIdleTimeout	The **PoFxSetDeviceIdleTimeout** routine specifies the minimum time interval from when the last component of the device enters the idle condition to when the power management framework (PoFx) calls the driver's *DevicePowerNotRequiredCallback* callback routine.
PoFxStartDevicePowerManagement	The **PoFxStartDevicePowerManagement** routine completes the registration of a device with the power management framework (PoFx) and starts device power management.
PoFxUnregisterDevice	The **PoFxUnregisterDevice** routine removes the registration of a device from the power management framework (PoFx).

Device Power Management Callbacks

These callback routines are required by the power management framework (PoFx) to enable device power management. The driver that is the power policy owner for the device implements these callback routines. PoFx calls these routines to query and configure the power states of the components in the device.

TOPIC	DESCRIPTION
ComponentActiveConditionCallback	The *ComponentActiveConditionCallback* callback routine notifies the driver that the specified component completed a transition from the idle condition to the active condition.
ComponentIdleConditionCallback	The *ComponentIdleConditionCallback* callback routine notifies the driver that the specified component completed a transition from the active condition to the idle condition.
ComponentIdleStateCallback	The *ComponentIdleStateCallback* callback routine notifies the driver of a pending change to the Fx power state of the specified component.
ComponentPerfStateCallback	The *ComponentPerfStateCallback* callback routine notifies the driver that its request to change the performance state of a component is complete.
DevicePowerNotRequiredCallback	The *DevicePowerNotRequiredCallback* callback routine notifies the device driver that the device is not required to stay in the D0 power state.
DevicePowerRequiredCallback	The *DevicePowerRequiredCallback* callback routine notifies the device driver that the device must enter and remain in the D0 power state.
PowerControlCallback	The *PowerControlCallback* callback routine performs a power control operation that is requested by the power management framework (PoFx).

Device Power Management Structures

The the power management framework (PoFx) defines these structures to support device power management.

TOPIC	DESCRIPTION
PO_FX_COMPONENT_V1 PO_FX_COMPONENT_V2	The **PO_FX_COMPONENT** structure describes the power state attributes of a component in a device.
PO_FX_COMPONENT_IDLE_STATE	The **PO_FX_COMPONENT_IDLE_STATE** structure specifies the attributes of an Fx power state of a component in a device.
PO_FX_COMPONENT_PERF_INFO	The **PO_FX_COMPONENT_PERF_INFO** structure describes all the sets of performance states for a single component within a device.
PO_FX_COMPONENT_PERF_SET	The **PO_FX_COMPONENT_PERF_SET** structure represents a set of performance states for a single component within a device.
PO_FX_DEVICE_V1 PO_FX_DEVICE_V2 PO_FX_DEVICE_V3	The **PO_FX_DEVICE** structure describes the power attributes of a device to the power management framework (PoFx).
PO_FX_PERF_STATE	The **PO_FX_PERF_STATE** structure represents a performance state for a single component within a device.

TOPIC	DESCRIPTION
PO_FX_PERF_STATE_CHANGE	The **PO_FX_PERF_STATE_CHANGE** structure contains information about a change to a performance state that is being requested by calling the PoFxIssueComponentPerfStateChange or PoFxIssueComponentPerfStateChangeMultiple routine.

Device Power Management Enumerations

The power management framework (PoFx) defines these enumerations to support device power management.

TOPIC	DESCRIPTION
PO_FX_PERF_STATE_TYPE	The **PO_FX_PERF_STATE_TYPE** enumeration contains values that describe the type of performance states in a PO_FX_COMPONENT_PERF_SET.
PO_FX_PERF_STATE_UNIT	The **PO_FX_PERF_STATE_UNIT** enumeration contains values that describe the type of unit that is controlled by the performance states in a PO_FX_COMPONENT_PERF_SET.

Device Power Management Constants

PO_FX_FLAG_XXX flag bits

The **PO_FX_FLAG_XXX** constants are flag bits that indicate whether a request to change the condition of component is performed synchronously or asynchronously.

```
#define PO_FX_FLAG_BLOCKING   0x1
#define PO_FX_FLAG_ASYNC_ONLY 0x2
```

PO_FX_FLAG_XXX constants

CONSTANT	DESCRIPTION
PO_FX_FLAG_BLOCKING	Make the condition change synchronous. If this flag is set, the routine that requests the condition change does not return control to the calling driver until the component hardware completes the transition to the new condition. This flag can be used only if the caller is running at IRQL < DISPATCH_LEVEL.
PO_FX_FLAG_ASYNC_ONLY	Make the condition change fully asynchronous. If this flag is set, the calling driver's callback routine is called from a thread other than the thread in which the routine that requests the condition change is called. Thus, the routine that requests the condition change always returns asynchronously without waiting for the callback to complete.

PO_FX_FLAG_XXX remarks

The Flags parameter to the following routines can be set to a **PO_FX_FLAG_XXX** constant:

- PoFxActivateComponent
- PoFxIdleComponent
- PoFxIssueComponentPerfStateChange
- PoFxIssueComponentPerfStateChangeMultiple

The **PO_FX_FLAG_BLOCKING** and **PO_FX_FLAG_ASYNC_ONLY** flag bits are mutually exclusive. The caller can set one or the other flag bit in the Flags parameter, but not both flag bits.

PO_FX_FLAG_XXX Requirements

VERSION	HEADER
Supported starting with Windows 8.	Wdm.h

PO_FX_FLAG_PERF_XXX flag bits

The **PO_FX_FLAG_PERF_XXX** constants are flag bits that define how the power management framework (PoFx) manages performance states for a device component.

```
#define PO_FX_FLAG_PERF_PEP_OPTIONAL   0x1
#define PO_FX_FLAG_PERF_QUERY_ON_F0 0x2
#define PO_FX_FLAG_PERF_QUERY_ON_ALL_IDLE_STATES 0x4
```

CONSTANT	VALUE	DESCRIPTION
PO_FX_FLAG_PERF_PEP_OPTIONAL	1 (0x1)	Indicates that the driver can change performance states without assistance from the platform extension plug-in (PEP), or that the driver is registering performance states with PoFx for logging purposes only. If this flag is set, the PoFxRegisterComponentPerfStates call will still succeeded if the PEP does not support performance states for the component.
PO_FX_FLAG_PERF_QUERY_ON_F0	2 (0x2)	For some devices, the PEP may need to a place a performance state set for a component into a certain performance state (known as a *nominal performance state*) when it idles the component. Drivers set this flag if the component contains nominal performance states, in which case PoFx will query the PEP to determine the current performance state when the component transitions to F0.
PO_FX_FLAG_PERF_QUERY_ON_ALL_IDLE_STATES	4 (0x4)	For some devices, the PEP may need to a place a performance state set for a component into a certain performance state (known as a *nominal performance state*) when it transitions the component between idle states. Drivers set this flag if this component contains nominal performance states, in which case PoFx will query the PEP to determine the current performance state when the component transitions between idle states.

PO_FX_FLAG_PERF_XXX remarks

The Flags parameter to the PoFxRegisterComponentPerfStates routine can be set to a **PO_FX_FLAG_PERF_XXX** constant.

PO_FX_FLAG_PERF_XXX requirements

REQUIREMENTS	VERSION
Supported starting with Windows 10.	Wdm.h

Component-Level Power Management

8/8/2019 • 6 minutes to read • Edit Online

Starting with Windows 8, the power management framework (PoFx) enables a driver to manage the power states of the individual components in a device. Component-level power management exists side-by-side with device-level power management.

Overview of Component-Level Power Management

Windows 7 and earlier versions of the operating system only provide support for device-level power management, which enables a driver to support D-states in a device. The Advanced Configuration and Power Interface (ACPI) specification defines device power states D0 (fully on) through D3 (fully off), and defines system power states S0 (fully on) through S5 (fully off). These versions of Windows do not provide mechanisms to independently manage the power supplied to the individual components in a device. In these versions of Windows, some drivers can implement custom power controls for components, but these controls typically add complexity to drivers, and might be feasible only if component power settings are controlled within the device.

Starting with Windows 8, PoFx adds support for component-level power management. This enables a driver to support some number of component power states, F0, F1, and so on, where F0 is the fully on state. All components support the F0 state. The driver that is the power policy owner (PPO) for the components in a device is responsible for defining any additional, low-power Fx power states that a component might have. (Typically, the function driver for a device is the PPO.) This driver determines the number of low-power Fx states per component and the attributes of each Fx state. The Fx states that this driver defines might vary from component to component in the same device.

PoFx provides a device driver interface (DDI) through which a driver can supply status and capabilities information about the components in a device. This information includes the current activity level of each component, the time required by the component to change from one power state to another, and the amount of latency that can be tolerated by clients of the device when the component wakes from a low-power state. In addition, PoFx obtains system-wide information about the power and clock domains to which the component belongs. (The devices in a particular power domain share a common power rail; the devices in a particular clock domain share a common clock signal.)

Based on this information, PoFx makes intelligent decisions about when a component should enter a low-power state and which low-power state to enter. The decision process involves information from other components and other devices, and takes into account the dependencies between the devices and components in the various power and clock domains.

Introduction to the PoFx API for Component-Level Power Management

To register a device to be managed by PoFx, the driver calls the **PoFxRegisterDevice** routine. The driver passes this routine a **PO_FX_DEVICE** structure that, among other data, contains an array of **PO_FX_COMPONENT** structures. Each element in this array describes the Fx power states of a component in the device and the attributes of each Fx state. (At minimum, a component that does not support component-level power management implements only the F0 state.) The attributes of a particular Fx power state in a particular component are described by a **PO_FX_COMPONENT_IDLE_STATE** structure, which contains the following values:

- The transition latency, which is the time required to make a transition from this Fx state to the F0 (fully on) state.

- The residency requirement, which is the time that a component must spend in this Fx state to make a transition to the state worthwhile.
- The nominal power, which is the power that is consumed by the component in this Fx state.

PoFx uses this information (in addition to other system-wide inputs and dependencies) to make intelligent decisions about which Fx power state a component should be in at any particular time. PoFx must balance two competing objectives. First, a component that is idle should be configured to consume as little power as possible. Second, a component must be prepared to switch from a low-power Fx state to F0 quickly enough to maintain the appearance of a device that is always on and always connected.

Component-level power management can be performed only when a device is in the D0 (fully on) power state. When a device is in the D1 (almost on), D2 (almost off), or D3 power state, the device is inaccessible. When the device is in the D0 state, only components that the driver is actively using need to remain in the F0 state. Idle components can potentially switch to low-power Fx states to reduce power consumption.

While a device is in the D0 power state, the driver follows a simple protocol to enable component-level power management. When the driver needs to access a component, the driver calls the **PoFxActivateComponent** routine to request access to the component. If the component is in a low-power Fx state when this call occurs, PoFx initiates a transition to the F0 state and notifies the driver when this transition is complete. The driver can then access the component. When the driver no longer needs to access the component, the driver calls the **PoFxIdleComponent** routine to notify PoFx. In response to this call, PoFx can potentially switch the component to a low-power Fx state.

A component that is accessible is in the *active condition*. A component that is inaccessible is in the *idle condition*. To track the accessibility of the components in a device, PoFx maintains an activation reference count on each component. A **PoFxActivateComponent** call increments the count on the specified component by one, and a **PoFxIdleComponent** call decrements the count by one.

If a **PoFxActivateComponent** call increments the count from zero to one, PoFx initiates a transition from the idle condition to the active condition, and notifies the driver when this transition completes. If a **PoFxActivateComponent** occurs when the component is already in the active condition, the component stays in the active condition and the driver receives no notification.

If a **PoFxIdleComponent** call decrements the count from one to zero, PoFx initiates a transition from the active condition to the idle condition, and notifies the driver when this transition is complete. If a **PoFxIdleComponent** call decrements the count but the count remains nonzero, the component stays in the active condition and the driver receives no notification.

The activation reference count conveniently handles situations in which two or more code paths in the same driver might need to concurrently access the same component in a device. By maintaining this count, PoFx enables the various parts of the driver to independently maintain access to the component without requiring the driver to centrally manage access to the component.

The active/idle condition of a component is the only reliable means for a driver to determine whether a component is accessible. A component that is in the F0 power state but is in the idle condition might be about to switch to a low-power Fx state.

A component that is in the active condition is always in the F0 state. The component cannot leave F0 until it enters the idle condition. A component that is in the idle condition might be in F0 or in a low-power Fx state. If a component is in a low-power Fx state when a **PoFxActivateComponent** call initiates a transition from the idle condition to the active condition, PoFx must first switch the component to F0 before the component can enter the active condition.

Related topics

Device Power Management Reference

Component-Level Performance State Management

6/25/2019 • 3 minutes to read • Edit Online

Starting with Windows 10, the power management framework (PoFx) enables a driver to define one or more sets of individually adjustable performance states for individual components within a device. The driver can use performance states to throttle a component's workload to provide just enough performance for its current needs.

Overview of Performance States

In Windows 8 and Windows 8.1, PoFx provides idle states (F-states) for component-level power savings by power and clock rail gating when a specific F-state is entered. This model saves power when a component is in an idle state (non-F0), but does not provide any mechanism for optimizing power usage or balancing it against performance needs when the component is active. Even though a component is active (in F0) and servicing a request, it may not require the full performance of the device. For example, a graphics card may need to only update a blinking cursor and this may not need full performance.

Variable performance states address this issue by allowing the driver to throttle a device's component to provide just enough performance for its current needs. In Windows 8 and Windows 8.1, if a component supports performance states, each driver must implement a proprietary performance state selection algorithm that is internal to the driver, and if needed, notify the platform extension plug-in (PEP) in a proprietary manner. The PEP is a software component that performs power management tasks that are specific to a particular product line of processor or System on a Chip (SoC) modules. Driver-specific proprietary performance state solutions have the disadvantage of being tightly coupled with the PEP, and cannot be easily debugged.

Starting with Windows 10, PoFx provides an API for performance state management. This API has two main goals:

- It provides a standard way for device drivers to notify the PEP about performance state changes so that the PEP may take the appropriate action.
- It provides a standard way for drivers to notify the OS of performance state changes for logging and analysis in Windows Performance Analyzer (WPA), without needing a custom plug-in for each driver.

Introduction to the PoFX API for Component-Level Performance States

PoFx enables a device to define the following types of performance states for each component:

- A discrete number of states in the units of frequency (measured in Hz), bandwidth (measured in bits per second), or an opaque index number.
- A continuous distribution of states between a minimum and maximum value.

Performance states are organized into sets and are registered on a per-component basis. Performance states within a set must increase monotonically. Most drivers are expected to define a single set of performance states per component. For example, a driver might define one set of performance states to control the clock frequency for a component. However, some drivers may need to define more than one performance state set to control multiple dimensions of performance states for a component. For example, a driver might define two sets of performance states to control the clock frequency and bus bandwidth.

To register a device component for performance state management by PoFx, a driver follows these general steps:

1. The driver registers the device components to be managed by PoFx. For more information, see Component-Level Power Management.
2. The driver registers support for performance states by calling **PoFxRegisterComponentPerfStates**. As

part of the registration call, drivers can either define a given component's performance state themselves or defer to the platform extension plug-in (PEP) to define them instead.

Either the device driver or the PEP must hold knowledge of the performance states, including the number of performance state sets per component, the type of performance state (discrete or range-based), and the details of the values and count of the actual performance states. If the PEP does not support performance states, the driver may still register for performance state support with PoFx and notify the OS of performance state changes for logging and analysis in Windows Performance Analyzer (WPA).

In either case, upon successful completion of **PoFxRegisterComponentPerfStates**, the driver has a **PO_FX_COMPONENT_PERF_INFO** structure that contains the registered performance state sets.

3. When the driver decides a component should change performance states, it calls **PoFxIssueComponentPerfStateChange** or **PoFxIssueComponentPerfStateChangeMultiple**. PoFx invokes the driver-provided **ComponentPerfStateCallback** routine when the performance state change is complete.

4. The driver is informed by the **ComponentPerfStateCallback** routine whether the PEP succeeded or denied the performance state change. If the PEP succeeded the change, the driver then performs whatever work it needs to take to change the performance state from its perspective. If the PEP denied the change, the driver may choose to do nothing or retry the request again with the same or an alternate performance state.

Related topics

Device Power Management Reference

Introduction to the Directed Power Management Framework

7/22/2019 • 3 minutes to read • Edit Online

Starting in Windows 10, version 1903, version 3 of the run-time power management framework (PoFx) provides an optional directed power model, Directed PoFx (DFx).

With DFx, the operating system directs device stacks to enter their appropriate low-power idle states when the system transitions to idle, and thereby enables the system to enter low power more reliably.

The objective is to make systems more power-efficient and to reduce energy consumption for Windows devices across form factors.

DFx currently supports D-state management only. DFx skips any device subtree with an F-state constraint.

Requirements for WDF (non-miniport) drivers

A WDF driver that specifies **SystemManagedIdleTimeout** or **SystemManagedIdleTimeoutWithHint** in the WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS structure can opt into DFx by adding the following registry key to the INF's AddReg directive section within the DDInstall.HW section:

```
HKR,"WDF","WdfDirectedPowerTransitionEnable",0x00010001,1
```

Because requesting system-managed idle timeout causes WDF to register with PoFx on the driver's behalf, the driver does not need to register with PoFx in this scenario.

If the driver specifies **DriverManagedIdleTimeout**, consider switching to system-managed idle timeout. If that is not feasible, use the guidelines in the WDM section below to opt into DFx.

If the WDF driver does not use runtime power management, add support for it and use system-managed idle timeout. To do so, provide an WDF_DEVICE_POWER_POLICY_IDLE_SETTINGS structure as input to **WdfDeviceAssignS0IdleSettings**.

Requirements for WDM (non-miniport) drivers

If your driver does not use the system-managed idle support provided by WDF (the driver is either a WDF driver using driver-managed idle, or a WDM driver), it can still get DFx support by registering itself with PoFx. In this scenario, the driver registers with PoFx by implementing:

- PO_FX_DIRECTED_POWER_DOWN_CALLBACK callback function
- PO_FX_DIRECTED_POWER_UP_CALLBACK callback function

Provide pointers to these callbacks in a PO_FX_DEVICE_V3 structure that is input to the **PoFxRegisterDevice** function.

To get DFx support, a driver must:

- Provide the `PO_FX_DIRECTED_POWER*` callbacks when registering for PoFx
- Call **PoFxReportDevicePoweredOn** from its PO_FX_DIRECTED_POWER_UP_CALLBACK callback function on resume from Sx transitions

Example

The following example shows the self-registration option described above:

```
PO_FX_DEVICE_V3 MyPoFxDevice;
POHANDLE MyPoFxHandle;

RtlZeroMemory(&MyPoFxDevice, sizeof(PO_FX_DEVICE_V3));
MyPoFxDevice.Version = PO_FX_VERSION_V3;

// Initialize other PoFx callbacks and other fields like
// components and their idle states.

MyPoFxDevice.DirectedPowerUpCallback = <Driver's DFx power up callback>
MyPoFxDevice.DirectedPowerDownCallback = <Driver's DFx power down callback>

Status = PoFxRegisterDevice(
  <Driver's device object>,
  (PPO_FX_DEVICE)&MyPoFxDevice,
  &MyPoFxHandle);
  if (!NT_SUCCESS(Status)) {
  return Status;
}
```

If your driver specified `PO_FX_VERSION_V1` previously, note that `PO_FX_DEVICE_V3` structures uses `PO_FX_COMPONENT_V2` for the component array structure.

Requirements for miniport drivers

For device classes that follow a port/miniport driver model, system-supplied port drivers typically handle power policy ownership. Most miniports are not expected to require any code changes to opt into DFx, as the corresponding port driver is expected to handle DFx support.

Testing

Microsoft provides three tests you can use for DFx: a single-device test in the Windows Driver Kit intended for testing user-specified devices, a device-level HLK test, and a system-level HLK test intended for testing all devices on a system.

The single-device test is available as part of the PwrTest tool that ships with the WDK. To access it, run the tool with the `/directedfx` switch. For more information, see PwrTest DirectedFx Scenario.

For information about HLK tests, please see the following pages:

- Directed FX Single Device Test
- Directed FX System Verification Test

Testing DFx after an S4 transition is recommended in order to catch any cases where a driver may not be correctly calling PoFxReportDevicePoweredOn after resume from S4.

DFx and S-state transitions

- The target D-state for DFx transitions should match that for Runtime D3 (RTD3), which may be different than the target D-state for S3/S4 transitions. Consider a scenario in which a device enters D2 for RTD3, but enters D3 for S3/S4. In this case, the target D-state for DFx should be D2.
- Similarly, the arm-for-wake behavior for DFx should match that for RTD3, which may differ from that used in S3/S4 transitions. For example, a device may enter D2/wake-armed for RTD3, but enter D3/no-wake-armed for S3/S4. In this scenario, DFx transitions should also enter D2/wake-armed.

DFx and Runtime D3 (RTD3)

- With RTD3, a device typically enters a lower power D-state when it goes idle. If new work arrives, the device immediately wakes to D0. With DFx, the device should continue to remain in its target D-state (and pend new work on its queues) until PoFx directs it to power back up.

See Also

- Prepare hardware for modern standby
- PwrTest
- PO_FX_DEVICE_V3 structure
- PO_FX_DIRECTED_POWER_DOWN_CALLBACK callback function
- PO_FX_DIRECTED_POWER_UP_CALLBACK callback function
- PoFxCompleteDirectedPowerDown function
- PwrTest DirectedFx Scenario
- Directed FX Single Device Test
- Directed FX System Verification Test

Platform Extension Plug-ins (PEPs)

12/5/2018 • 2 minutes to read • Edit Online

Starting with Windows 10, the run-time power management framework (PoFx) supports platform extension plug-ins (PEPs) for enhanced device component management.

In this section

TOPIC	DESCRIPTION
Using PEPs for ACPI services	This topic provides information about using platform extension plug-ins (PEPs) for ACPI services.
Platform Performance Thresholds	There are two types of performance thresholds - static thresholds which remain fixed for the platform and dynamic thresholds that change at runtime. This topic describes the static performance thresholds of the platform and the allowed range for the dynamic threshold.

Using PEPs for ACPI services

6/25/2019 • 3 minutes to read • Edit Online

This topic provides information about using platform extension plug-ins (PEPs) for ACPI services.

PEPs provide dynamic, runtime ACPI methods. The static tables (FADT, MADT, DBG2, etc.) must be implemented in the ACPI firmware, as well as the DSDT/SSDT device hierarchy.

PEPs are intended to be used for off-SoC power management methods. Since they are installable binaries, they can be updated on-the-fly as opposed to ACPI firmware which requires a firmware flash. This means you could improve your power management code on platforms that you've already shipped by posting an updated driver on Windows Update. Power management was the original intent for PEPs, but they can be used to provide or override any arbitrary ACPI runtime method.

PEPs play no role in the construction of the ACPI namespace hierarchy because the namespace hierarchy must be provided in the firmware DSDT. When the ACPI driver evaluates a method at runtime, it will check against the PEP's implemented methods for the device in question, and, if present, it will execute the PEP and ignore the firmware's version. However, the device itself must be defined in the firmware.

Providing power management using PEPs can be much easier to debug than code written for the ACPI firmware because of the tools available. Tools for debugging ACPI firmware are unfamiliar to most and tool options are limited. In contrast, PEPs are developed as Windows drivers so developers can use whatever development and debugging tools they are most comfortable with.

When using a PEP in place of an ACPI service, no special action or operation is needed in order to claim the role of the service. When a method is implemented in the PEP, Windows will use it automatically. If a firmware version of the same method on the same device is provided, it will be ignored.

PEPs are loaded very early so that their services are available for the device driver. Additionally, the abstraction layer through Windows is designed to be transparent to device drivers. The driver should expect to be able to interact with its ACPI methods as if a PEP weren't in use.

When using PEP for both device power management (DPM) and ACPI services, it's advisable to use separate PEP handles, but this is only a matter of preference. When sharing the handle DPM and ACPI state can be tracked easily for a device because the handle is the same. However, handle lifetime management is a little more complicated. The PEP will need to provide reference counting for the handle to make sure it is only deleted after both DPM and ACPI services have been torn down for that handle (i.e., both **PEP_DPM_UNREGISTER_DEVICE** and **PEP_NOTIFY_ACPI_UNREGISTER_DEVICE** have been received on that handle). When different handles are used, DPM and ACPI state will be tracked separately, but handle lifetime management is simpler. In this case, the handle can be destroyed when the corresponding unregister notification is sent.

To simplify the process of working with ACPI resources, the power management framework (PoFx) provides the PEP_REQUEST_COMMON_ACPI_CONVERT_TO_BIOS_RESOURCES helper routine to convert ACPI resources to BIOS resources.

PEPs are responsible for scheduling work that cannot be performed synchronously in response to an ACPI notification from PoFx but the method used is determined by the PEP developer. Typically, the PEP will queue the work on some internal queue and then start a worker thread if needed. It is also possible that the work needs to wait for some external event (e.g. device interrupt) and will be processed in the context of that event. Once the work is done, a PEP can request PoFx to query for work by invoking **PEP_KERNEL_INFORMATION_STRUCT_V3**->*RequestWorker*(). In response, PoFx will send a **PEP_DPM_WORK notification** for PEPs that implement the DPM notification handler

(*AcceptDeviceNotification*) or a **PEP_NOTIFY_ACPI_WORK notification** for PEPs that implement the ACPI-only notification handler (*AcceptAcpiNotification*).

Related topics

ACPI system description tables
PEP_DPM_UNREGISTER_DEVICE
PEP_NOTIFY_ACPI_UNREGISTER_DEVICE
PEP_KERNEL_INFORMATION_STRUCT_V3
PEP_DPM_WORK
PEP_NOTIFY_ACPI_WORK
RequestWorker
AcceptDeviceNotification
ACPI notifications

Platform Performance Thresholds

12/5/2018 • 2 minutes to read • Edit Online

There are two types of performance thresholds: static thresholds which remain fixed for the platform and dynamic thresholds that change at runtime. This topic describes the static performance thresholds of the platform and the allowed range for the dynamic threshold.

The static performance thresholds have the following definitions:

Highest performance
The absolute maximum performance an individual processor may reach, assuming ideal conditions. This performance level may not be sustainable for long durations, and may only be achievable if other platform components are in a specific state (for example, it may require other processors be in an idle state).

Nominal performance
The maximum sustained performance level of the processor, assuming ideal environmental conditions (i.e. optimal ambient temperature, the processor is not already hot due to prior activity, available current is not restricted due to a low/cold battery). All processors are expected to be able to sustain continuous activity at their nominal performance simultaneously for at least one second.

Lowest Nonlinear Performance
The lowest performance level at which nonlinear power saving is achieved as performance is scaled. For example, due to the combined effects of voltage and frequency scaling better than liner power saving can be achieved by running at a lower performance state. Above this threshold, lower performance levels should be more energy efficient than higher performance levels.

Lowest Performance
The absolute lowest performance level of the platform. Selecting a performance level lower than the lowest nonlinear performance level may be equivalent from an efficiency perspective or may actually cause an efficiency penalty, but should reduce the instantaneous power consumption of the processor.

Note All static performance levels do not need to be distinct. A platform's nominal performance level may also be its highest performance level, for example.

The platform may optionally also express a dynamic performance threshold, the *Guaranteed Performance* threshold. If present, this represents the maximum sustained performance level of a processor, taking into account all known external constraints (power budgeting, thermal constraints, power source, etc.). All processors are expected to be able to sustain their guaranteed performance levels simultaneously for at least one second. The guaranteed performance level is required to fall in the range [Lowest Performance, Nominal performance], inclusive.

Heterogeneous Performance Thresholds

The PEP must use the same performance scale for all processors in the system. On platforms with heterogeneous processors, the performance characteristics of all processors may not be identical. In this case, the PEP must synthesize a performance scale that adjusts for differences in processors, such that any two processors running the same workload at the same performance level will complete in approximately the same time. The PEP should expose different capabilities for different classes of processors, so as to accurately reflect the performance characteristics of each processor.

Supporting Devices that Have Wake-Up Capabilities

6/25/2019 • 2 minutes to read • Edit Online

Drivers for devices that can respond to external wake signals must be able to handle **IRP_MN_WAIT_WAKE** requests (*wait/wake IRPs*). The power policy owner for such a device must be able to send an **IRP_MN_WAIT_WAKE** request.

Typically, whatever causes the device to assert the wake signal is also a normal service event for the device. For example, user input, which might cause a keyboard to wake up the system, is a normal event for the keyboard and its drivers.

The first topic of this section, Overview of Wait/Wake Operation, contains information useful in writing any driver. The following additional topics provide detailed information about handling and sending wait/wake IRPs:

Receiving a Wait/Wake IRP

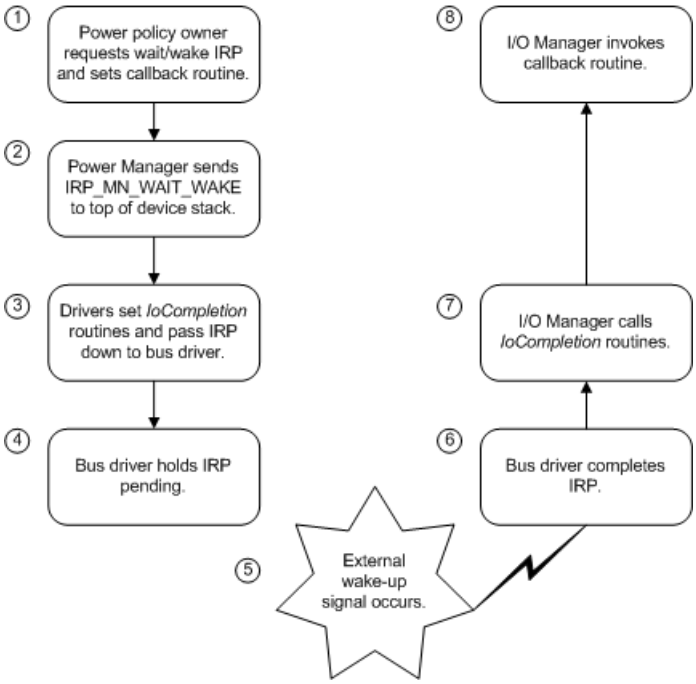
Sending a Wait/Wake IRP

Canceling a Wait/Wake IRP

Overview of Wait/Wake Operation

6/25/2019 • 2 minutes to read • Edit Online

The operating system's wake-up mechanism works as shown in the following figure.



1. While the system and device are in the working state, the power policy owner for a device determines that its device should be enabled ("armed") for wake-up. The power policy owner requests a power IRP (**PoRequestPowerIrp** with minor code **IRP_MN_WAIT_WAKE**) to be sent to its PDO to inform all drivers in its device stack. In the request, the policy owner specifies a callback routine (not the same as an *IoCompletion* routine).
2. The power manager, through the I/O manager, sends the IRP to the top of the device stack.
3. Drivers set *IoCompletion* routines and pass the IRP down until it reaches the bus driver.
4. The bus driver enables wake-up on the physical device, if it can, and marks the IRP pending. If necessary, it also requests a wait/wake IRP for its parent.
5. Sometime later, an external wake-up signal arrives.
6. The bus driver completes the **IRP_MN_WAIT_WAKE**.
7. I/O manager calls *IoCompletion* routines that were set as drivers passed the IRP down the stack.
8. I/O manager calls the callback routine set by the policy owner when it requested the IRP.

The **IRP_MN_WAIT_WAKE** request does not change the power state of the device or the system. It merely enables wake-up on the device so that later, if the device enters an appropriate sleep state, an external signal will cause the device (and possibly the system) to awaken.

When a wake-up signal arrives, the drivers' behavior is the same whether the device wakes the system or only itself. If the device is enabled for wake-up and the system is in a sleep state from which the device can awaken it, the device will awaken the system. If the device is enabled for wake-up and the system is in the working state, only the device will awaken.

Because computers and devices vary in design, particularly with respect to power planes, the supported system and device power states -- and thus the states that can support wait/wake -- are not the same on all hardware configurations. Therefore, any driver that owns power policy for its device and every bus driver must pay careful attention to the capabilities of the individual configuration on which it is running. For further information, see Determining Whether a Device Can Wake the System.

For further details on wait/wake operations, see Understanding the Path of Wait/Wake IRPs through a Device Tree and Overview of Wait/Wake IRP Completion.

Determining Whether a Device Can Wake the System

6/25/2019 • 2 minutes to read • Edit Online

Some devices, such as keyboards, modems, and network cards, can respond to external signals while in a device sleep state. As part of its power management technology, the operating system provides a way for such devices to wake a sleeping system, which can then restore its previous context. The software wake-up mechanism allows a system to awaken from any state except S5 (**PowerSystemShutdown**), depending on support in the system and device hardware and BIOS. A system in state S5 must always be rebooted.

Although the operating system is designed to awaken from any of the intermediate sleep states, the exact wake-up capabilities vary from computer to computer and device to device. Not all computers support all system sleep states; therefore, the ability to wake from certain states is meaningless on some computers.

Similarly, most devices neither support all device power states (D0 through D3) nor support wake-up from all the device power states that they do support.

The sleep states that a device can enter, along with the states from which it supports wake up, are described at enumeration by the bus driver and are stored in the **DEVICE_CAPABILITIES** structure. The following table lists the members of this structure that are relevant to wait/wake support.

MEMBER	DESCRIPTION
DeviceD1	True if device supports state PowerDeviceD1.
DeviceD2	True if device supports state PowerDeviceD2.
WakeFromD0	True if device can wake from PowerDeviceD0.
WakeFromD1	True if device can wake from PowerDeviceD1.
WakeFromD2	True if device can wake from PowerDeviceD2.
WakeFromD3	True if device can wake from PowerDeviceD3.
DeviceState [PowerSystemMaximum]	Specifies highest device power state that this device can support for each system power state, from PowerSystemUnspecified to PowerSystemShutdown.
SystemWake	Specifies lowest system power state (S0 through S4) from which the system can be awakened.
DeviceWake	Specifies lowest device power state (D0 through D3) from which the device can awaken.

The **DeviceWake** entry lists the lowest device power state from which the device can respond to a wake-up signal.

The value PowerDeviceUnspecified indicates that the device cannot wake the system. The **SystemWake** entry lists the lowest system power state from which the system can be awakened. These values are based on the capabilities of the parent devnode and drivers should not change them. For more information, see Reporting Device Power Capabilities.

In general, a device can wake the system if the following are true:

- The device is in a power state equal to or more-powered than the **DeviceWake** value.
- The system is in a power state equal to or more powered than the **SystemWake** value.

Understanding the Path of Wait/Wake IRPs through a Device Tree

6/25/2019 • 7 minutes to read • Edit Online

Within a single device stack, the power policy owner sends a wait/wake IRP and all drivers handle the wait/wake IRP, as outlined in Overview of Wait/Wake Operation and detailed in Sending a Wait/Wake IRP and Receiving a Wait/Wake IRP, respectively.

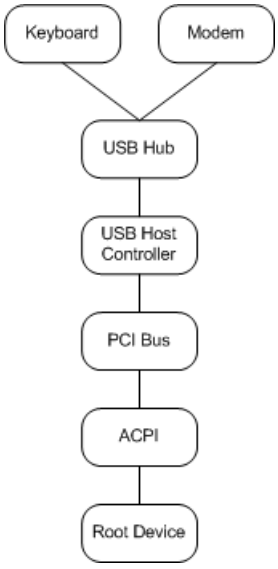
Within a branch of the device tree (which comprises a leaf devnode and the devnodes of its parents, grandparents, and so forth), drivers must cooperate to ensure that a wait/wake IRP reaches a driver that can enable all the necessary hardware for wake-up.

On ACPI computers, ACPI is responsible for enabling the system-specific General Purpose Event (GPE) register associated with the wake-up signal from each leaf device. Consequently, drivers must request and forward wait/wake IRPs until one reaches either an ACPI filter driver (inserted in the device stack at start-up) or the underlying Windows ACPI driver, Acpi.sys. In response, ACPI enables the register, holds the IRP pending until the signal arrives, and then completes the IRP. Because ACPI can respond to the wake-up signal, it does not forward the IRP to a lower driver.

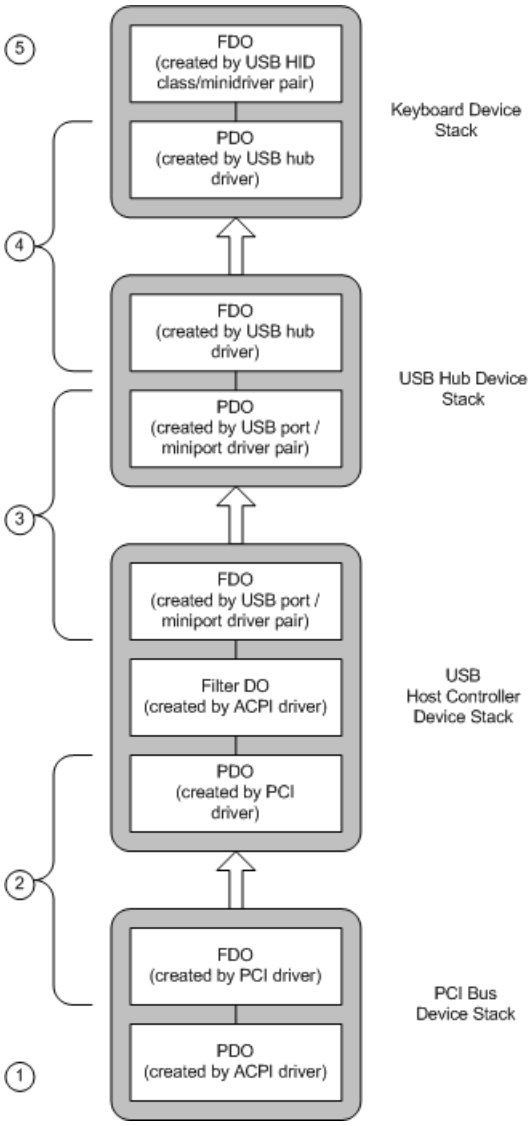
ACPI filter drivers, like the underlying ACPI driver itself, are transparent to other drivers. To provide maximum flexibility in hardware design, the exact position of an ACPI filter driver in any device stack is device- and system-specific. In designing a driver, you cannot make any assumptions about the presence or position of an ACPI filter in the device stack.

Keep in mind that drivers that enumerate children create a PDO for each child device and an FDO for the parent device. The driver thus acts as the bus driver for a child device and the function driver/policy owner for a parent device. Therefore, whenever a bus driver receives a wait/wake IRP for a child PDO, it should request another wait/wake IRP for its parent PDO.

The following figure shows a sample configuration in which such a situation occurs.



In the sample configuration, the keyboard and modem are children of the USB hub, which in turn is a child of the USB host controller, which is enumerated by the PCI bus. The following figure shows the device stacks for the keyboard in the sample configuration.

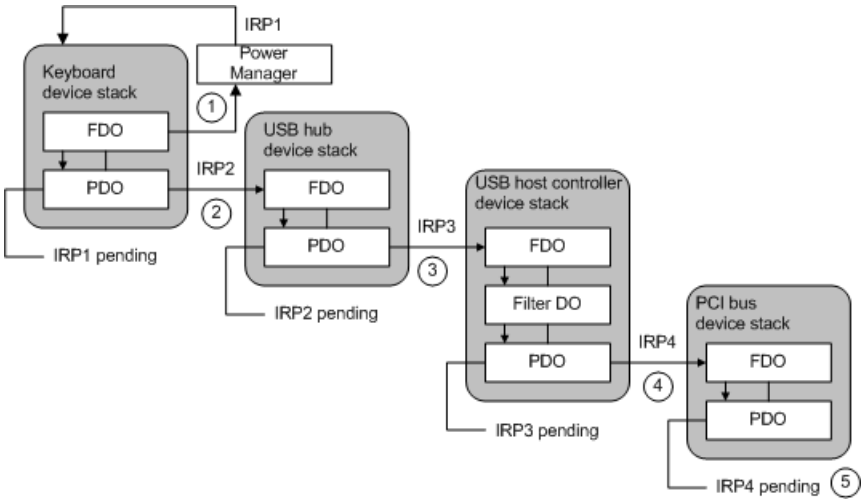


As the previous figure shows, reading from the bottom up:

1. The Windows ACPI driver, Acpi.sys, creates the PDO for PCI.
2. The PCI driver creates the PCI FDO and the USB host controller PDO and owns policy for the PCI device stack.
3. The USB host controller driver (a host port/miniport driver pair) creates the USB Host Controller FDO and the USB Hub PDO. It owns policy for the USB host controller device stack. Note that Acpi.sys creates a filter DO in this stack as well.
4. The USB hub driver creates the USB Hub FDO and the keyboard PDO. This driver owns power policy for the USB hub device stack.
5. The function driver for the keyboard is the USB HID class driver/minidriver pair. This driver creates the FDO for the keyboard and owns its power policy. Because the keyboard has no child devices, this driver creates no PDOs.

Note that each device stack might include additional optional filter DOs that are not shown.

To allow keyboard input to awaken the system, the policy owner for the keyboard requests an **IRP_MN_WAIT_WAKE** for its PDO. That IRP sets off a chain of other wait/wake IRPs, as shown in the following figure.



When a bus driver receives an **IRP_MN_WAIT_WAKE** targeted to a PDO it created, it must request another **IRP_MN_WAIT_WAKE** for the device stack for which it owns power policy and created an FDO.

As the previous figure shows:

1. The keyboard driver calls **PoRequestPowerIrp** to send a wait/wake IRP (IRP1) to its PDO.

   The power manager allocates the IRP and sends it through the I/O manager to the top of the device stack for the keyboard. Drivers set *IoCompletion* routines and pass the IRP down the stack until it reaches the keyboard PDO. The USB hub driver, which acts as the bus driver for the keyboard, holds IRP1 pending.

2. Because the USB hub driver cannot wake the system when the wake-up signal arrives, the USB hub driver must call **PoRequestPowerIrp** to request a wait/wake IRP (IRP2) for the USB hub device stack.

   The power manager sends this IRP to the top of the USB hub device stack. The drivers in this stack set *IoCompletion* routines and pass the IRP down to the USB host controller driver (which acts as the bus driver for the USB hub). The USB host controller driver holds IRP2 pending until the keyboard signals a wake event.

3. Similarly, the USB host controller driver cannot wake the system, so the USB host controller driver calls **PoRequestPowerIrp** to send a wait/wake IRP (IRP3) to the USB host controller device stack.

   The power manager sends this IRP to the top of the USB host controller device stack, where drivers set *IoCompletion* routines and pass the IRP down to the PCI driver (which acts as the bus driver for the USB hub). The PCI driver holds IRP3 pending until the keyboard signals a wake event.

4. The PCI driver cannot wake the system, so the PCI driver calls **PoRequestPowerIrp** to send a wait/wake IRP (IRP4) to the PCI device stack. Its parent is the root device, for which ACPI is the bus driver.

   The power manager sends the IRP to the top of the PCI bus device stack; its drivers set completion routines and pass the IRP down to the Windows ACPI driver, Acpi.sys.

5. Acpi.sys can wake the system, so it does not send a wait/wake IRP to any other PDO. Acpi.sys holds IRP4 pending until a wake signal arrives.

When the keyboard asserts the wake-up signal, Acpi.sys intercepts it. ACPI, however, cannot determine that the keyboard asserted the signal, only that the signal came through the root device. Acpi.sys then completes IRP4, and the I/O manager calls *IoCompletion* routines traveling back up the PCI device stack. When IRP4 is complete and all *IoCompletion* routines have run, the PCI driver's callback routine is invoked. In its callback routine, the PCI driver determines that the signal came through the USB host controller. The PCI driver then completes IRP3. The same sequence occurs through the USB host controller stack and the USB hub stack, until the keyboard driver receives IRP1. At this point, the keyboard driver can service the wake-up event, as necessary.

Each time a driver sends a wait/wake IRP to a parent PDO, it must set a *Cancel* routine for its own IRP. Setting a

Cancel routine gives the driver an opportunity to cancel the new IRP if the IRP that triggered it is canceled. In the USB example, if the keyboard driver cancels its wait/wake IRP (thus disabling keyboard wake-up), the USB hub, USB host controller, and PCI drivers must cancel the IRPs that they sent as a consequence of the keyboard IRP. For more information, see Cancel Routines for Wait/Wake IRPs.

Although a parent driver might enumerate more than one child that can be enabled for wait/wake, only one wait/wake IRP can be pending for a PDO. In such cases, the parent driver should make sure that it keeps a wait/wake IRP pending whenever any of its devices is enabled for wake-up. To do so, the driver increments an internal counter each time it receives a wait/wake IRP. Each time the driver completes a wait/wake IRP, it decrements the count and, if the resulting value is nonzero, sends another wait/wake IRP to its device stack.

For example, in the USB configuration shown previously in the Sample USB Configuration figure, the USB hub enumerates two devices, a keyboard and a modem. When the USB hub driver receives a wait/wake IRP for the keyboard PDO, it increments a count of wait/wake IRPs before requesting an IRP for its own PDO. If the modem's policy owner later enables wake-up for the modem, the USB hub driver pends the new IRP for the modem PDO and increments its wait/wake reference count. However, because the USB hub PDO cannot have two simultaneously pending wait/wake IRPs, the USB hub driver does not request a new wait/wake IRP for the USB hub PDO.

When a wake-up signal arrives from either the keyboard or modem, the USB hub driver determines which device signaled, completes the corresponding IRP, and decrements its reference count. Because both devices were enabled for wake-up (and thus its reference count is nonzero), it must send its own device stack another wait/wake IRP to "rearm" its own PDO for wake-up. (The same is true of the USB host controller and PCI driver.)

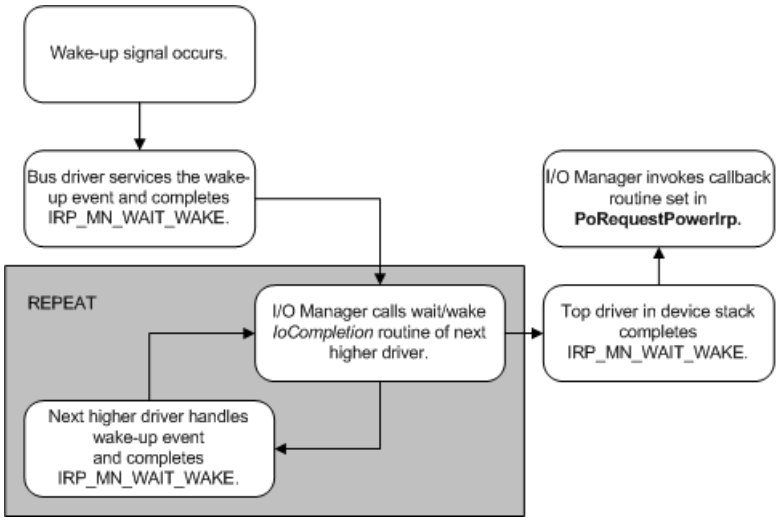
A driver does not, however, send itself an IRP to reenable wait/wake on the same device on which a wake-up signal just arrived. Only the device power policy manager can do that. Reenabling wait/wake is not automatic.

Overview of Wait/Wake IRP Completion

6/25/2019 • 2 minutes to read • Edit Online

A wait/wake IRP completes when a wake-up signal arrives. The wake-up signal is device-specific but is generally a normal service event for the device. For example, an incoming ring might cause a sleeping modem to awaken.

The following figure shows the steps in completing a wait/wake IRP.



When the signal occurs, control re-enters the bus driver at the point where the bus detects that the device has awakened. The bus driver services the event as required and calls **IoCompleteRequest** to complete the **IRP_MN_WAIT_WAKE** IRP for its PDO.

The I/O manager then calls the *IoCompletion* routine set by the next-higher driver in the device stack. In the *IoCompletion* routine, that driver services the wake-up signal as necessary and calls **IoCompleteRequest** to complete the IRP. The I/O manager continues to call *IoCompletion* routines working back up the device stack until all drivers have completed the IRP.

In its *IoCompletion* routine, any driver that enumerates more than one child device (creates more than one PDO) and has received wait/wake requests from more than one such device must send itself a wait/wake IRP to re-arm itself for wait/wake on another child. For details, see Understanding the Path of Wait/Wake IRPs through a Device Tree.

After calling *IoCompletion* routines set by drivers as they passed the IRP down the stack, the I/O manager invokes the callback routine set by the power policy owner when it requested the wait/wake IRP. In the callback routine, the policy owner should return its device to the working state and complete a pending wait/wake IRP for its child's PDO, if any.

Completing the child's IRP causes the I/O manager to call *IoCompletion* routines set by drivers in the child's device stack, and so on. Eventually, the policy owner that started the original wait/wake IRP on the devnode determines that its device asserted the wake-up signal, and all the pending wait/wake IRPs will be complete.

Receiving a Wait/Wake IRP

6/25/2019 • 2 minutes to read • Edit Online

All PnP drivers must be prepared to receive power IRPs with minor IRP code **IRP_MN_WAIT_WAKE**. How a driver handles a wait/wake IRP depends on its position in the device stack, the type of device(s) it controls, and the specific states from which its device supports wake-up.

The topics in this section provide guidelines for handling this IRP based on the type of driver and its level of wait/wake support.

Handling a Wait/Wake IRP in a Function (FDO) or Filter Driver (Filter DO)

6/25/2019 • 2 minutes to read • Edit Online

When a driver that creates an FDO or filter DO receives an **IRP_MN_WAIT_WAKE** request for the associated PDO, it can either simply pass the IRP down to the next-lower driver or take certain actions before passing down the IRP.

For Devices That Support Wake-Up

Upon receiving a wait/wake IRP, a function or filter driver should take the following steps:

1. Call **IoAcquireRemoveLock**, passing the current IRP, to ensure that the driver does not receive a PnP **IRP_MN_REMOVE_DEVICE** request while handling the wait/wake IRP.

   If **IoAcquireRemoveLock** returns a failure status, the driver should not continue processing the IRP. Instead, it completes the IRP (**IoCompleteRequest**), and return the failure status.

2. Inspect the value at **Irp->Parameters.WaitWake.PowerState** and compare the current device power state with **DeviceState**[SystemWake] in the **DEVICE_CAPABILITIES** structure.

   If the device supports wake-up, but not from the specified SystemWake state or not from the current device power state, the driver should fail the IRP as follows:

   - Set STATUS_INVALID_DEVICE_STATE in **Irp->IoStatus.Status**.
   - Complete the IRP (**IoCompleteRequest**), specifying a priority boost of IO_NO_INCREMENT.
   - Return the status set in **Irp->IoStatus.Status** from the *DispatchPower* routine.

3. Otherwise, set an *IoCompletion* routine for the IRP using **IoSetCompletionRoutine**. The *IoCompletion* routine should perform whatever tasks the driver requires to return the device to the working state.

   The *IoCompletion* routine will also be called if the IRP is canceled.

4. Save any information the driver might need in its *IoCompletion* routine.

5. Call **IoCallDriver** (in Windows 7 and Windows Vista) or **PoCallDriver** (in Windows Server 003, Windows XP, and Windows 2000), to pass the wait/wake IRP to the next-lower driver.

6. Call **IoReleaseRemoveLock** to release the previously acquired lock.

7. Return STATUS_PENDING from the *DispatchPower* routine. The driver must not change the value in **Irp->IoStatus.Status** while it holds the IRP.

For Devices That Do Not Support Wake-Up

If a function or filter driver receives a wait/wake IRP for a device that does not support wake-up, the driver should fail the IRP as follows:

1. Complete the IRP (**IoCompleteRequest**), specifying a priority boost of IO_NO_INCREMENT.

2. Return the status set in **Irp->IoStatus.Status** from the *DispatchPower* routine.

Handling a Wait/Wake IRP in a Bus Driver (PDO)

6/25/2019 • 2 minutes to read • Edit Online

Like other power IRPs, each wait/wake IRP must be passed all the way down the device stack to the bus driver (PDO), which is ultimately responsible for completing the IRP. Upon receiving the IRP, the bus driver can either fail it immediately or hold it pending for later completion. The following are the steps the bus driver must take:

1. Inspect the value at **Irp->Parameters.WaitWake.PowerState**. If the device supports wake-up, but not from the specified **SystemWake** state or not from the current device power state, the driver should fail the IRP as follows:
   - Set STATUS_INVALID_DEVICE_STATE in **Irp->IoStatus.Status**.
   - Complete the IRP (**IoCompleteRequest**), specifying a priority boost of IO_NO_INCREMENT.
   - Return the status set in **Irp->IoStatus.Status** from the *DispatchPower* routine.
2. Check whether a wait/wake IRP is already pending for the PDO. If so, set **Irp->IoStatus.Status** to STATUS_DEVICE_BUSY, increment the driver's internal count of wait/wake IRPs, and complete the IRP as described in the previous step.

   Only one wait/wake IRP can be pending for a PDO.
3. If the device supports wake-up from the specified system power state and no wait/wake IRP is already pending, call **IoMarkIrpPending** to indicate to the I/O manager that the IRP will be completed or canceled later. Do not set an *IoCompletion* routine.
4. Set the device hardware to enable wake-up.

   The specific mechanism by which a bus driver enables its hardware for wake-up is device-dependent. For a PCI device, Pci.sys is responsible for setting the PME-enable bit because this driver owns the PME register. For other devices, refer to the device-class-specific documentation.
5. If the PDO is the child of an FDO, request a wait/wake IRP for the FDO, making sure to set a *Cancel* routine for the current IRP (the IRP that it holds pending). Do not attempt to pass on or reuse the current IRP.
6. Return STATUS_PENDING from the *DispatchPower* routine.
7. When a wake-up signal arrives, call **IoCompleteRequest** to complete the pending wait/wake IRP, setting **Irp-IoStatus.Status** to STATUS_SUCCESS, and specifying a priority boost of IO_NO_INCREMENT.

For Devices That Do Not Support Wake-Up

If the device does not support wake-up, the bus driver (PDO) should proceed as follows:

1. Complete the wait/wake IRP by calling **IoCompleteRequest**, specifying IO_NO_INCREMENT.
2. Return from the *DispatchPower* routine, passing the value at **Irp->IoStatus.Status** as its return value.

IoCompletion Routines for Wait/Wake IRPs

6/25/2019 • 2 minutes to read • Edit Online

The I/O manager calls a driver's wait/wake *IoCompletion* routine after the next-lower driver in the device stack has completed the wait/wake IRP. Each function and filter (FDO) driver that handles a wait/wake IRP should set an *IoCompletion* routine for the IRP.

Each function or filter driver sets an *IoCompletion* routine as it handles the wait/wake IRP on its way down the device stack. The device power policy owner, which sends the IRP, might therefore have an *IoCompletion* routine in addition to a callback routine. Keep in mind that the callback routine is invoked after the *IoCompletion* routine and that the two have different requirements. For more information, see Wait/Wake Callback Routines.

The actions required in a wait/wake *IoCompletion* routine depend on the device and the type of driver. The following are some actions a driver might need to perform in its wait/wake *IoCompletion* routine:

1. Reset any relevant fields in the device extension. For example, when a wait/wake IRP is pending, most drivers set a flag and keep a pointer to the IRP in the device extension.
2. Reset the *Cancel* routine, if any, for the IRP by calling **IoSetCancelRoutine**, specifying a **NULL** pointer for the routine.
3. Call **IoCompleteRequest**, specifying IO_NO_INCREMENT, to complete the IRP.

As each successive driver completes the IRP, the I/O manager passes control to the *IoCompletion* routine of the next-higher driver going back up the stack.

After calling the *IoCompletion* routines set by drivers as they passed the wait/wake IRP down the device stack, the I/O manager calls the callback routine passed to **PoRequestPowerIrp** by the driver that sent the IRP. For further information, see Wait/Wake Callback Routines.

Sending a Wait/Wake IRP

6/25/2019 • 2 minutes to read • Edit Online

The minor power IRP code **IRP_MN_WAIT_WAKE** provides for waking a device or waking the system. Drivers of devices that can wake themselves or the system send **IRP_MN_WAIT_WAKE** requests. The system sends **IRP_MN_WAIT_WAKE** requests only to devices that always wake the system, such as the power-on switch.

A driver sends an **IRP_MN_WAIT_WAKE** request for one of two reasons:

1. Its device must be able to return to the working state from a sleep state in response to an external wake-up signal.

   For example, a modem's driver might send it a wait/wake IRP before setting it in power state D1 to conserve energy. The wait/wake IRP enables the modem to respond to an incoming call.

2. Its device must be able to wake the system in response to a wake-up signal.

   When the system goes to sleep, the modem might remain in state D1 with an **IRP_MN_WAIT_WAKE** pending. In this case, an incoming call would wake the system as well as the modem.

Whether a device is prepared to wake itself or the system, the actions its drivers must take are the same. The primary difference lies in how the device and system hardware respond to the initial wake-up signal. Driver behavior is the same in either case.

Determining When to Send a Wait/Wake IRP

6/25/2019 • 2 minutes to read • Edit Online

The driver that owns device power policy sends wait/wake IRPs on behalf of its device. Such a driver must send a wait/wake IRP when one of the following occurs:

- The driver is putting the device to sleep but the device must be able to wake up in response to an external wake-up signal.
- The system is going to sleep and the device must be able to awaken it.

The power policy owner should send the wait/wake IRP before any such conditions are imminent. It can send the IRP any time its device is in D0, but it must not send such an IRP while it is handling another set-power or query-power IRP. As a general rule, the driver should send the IRP during its handling of the Plug and Play manager's **IRP_MN_START_DEVICE** request, after it has initialized and started the device.

Wait/Wake IRP Requests

6/25/2019 • 2 minutes to read • Edit Online

To send an **IRP_MN_WAIT_WAKE**, a driver calls **PoRequestPowerIrp**, passing (among other parameters) a pointer to the target PDO, a system power state, and a pointer to a callback routine.

The system power state specifies the least-powered state from which this IRP can wake the system. The value must be equal to or more powered than the **SystemWake** state in the **DEVICE_CAPABILITIES** structure. For example, if a driver passes **PowerSystemSleeping2** in the IRP, the associated IRP could cause the system to wake from states S0, S1, and S2. In such a case, the system must support S0 and S2 (the highest- and lowest-powered states in the range) but need not support S1.

Every driver that requests a wait/wake IRP should specify a callback routine, which is invoked after all other drivers have completed the IRP. In this routine, the driver can do whatever is necessary to return its device to the working state.

In response to **PoRequestPowerIrp**, the power manager allocates a power IRP with minor code **IRP_MN_WAIT_WAKE** and sends it to the top of the device stack for the target PDO. The caller is returned a pointer to the allocated IRP, which it can use later if it has to cancel the IRP.

If no errors occur, **PoRequestPowerIrp** returns STATUS_PENDING. This status means that the IRP has been sent successfully and is pending completion.

A wait/wake IRP does not change the power state of the system or of a device. It simply enables a device's wake-up signal. The IRP remains pending until an external signal causes the system or device to awaken.

Wait/Wake Callback Routines

6/25/2019 • 2 minutes to read • Edit Online

When a driver requests a wait/wake IRP, it must specify a callback routine so that it can return the device to the working state (D0) when the wake-up event occurs. After the wake-up event occurs and all drivers have completed the IRP, the system calls the callback routine passed to **PoRequestPowerIrp**.

Because this callback routine is set on behalf of the driver that originated the IRP—and not for a driver that is handling the IRP—it must not call **PoStartNextPowerIrp**; only the *IoCompletion* routines set as drivers pass the IRP down the stack should start the next power IRP. Keep in mind that the policy owner not only sends the IRP but handles it, and it therefore might set an *IoCompletion* routine as it passes the IRP down the stack in addition to setting a callback routine when it requests the wait/wake IRP.

The callback routine has the following responsibilities:

1. If the driver controls more than one device, determine which of its devices signaled the wake-up.
2. Service the event that caused the wake-up signal.
3. Set the device that signaled the wake-up in the D0 state by calling **PoRequestPowerIrp** to send a **PowerDeviceD0** request. The driver must also call **PoSetPowerState** to inform the power manager of the new device power state. For more information, see Sending IRP_MN_QUERY_POWER or IRP_MN_SET_POWER for Device Power States.
4. If the driver set a *Cancel* routine for the IRP, call **IoSetCancelRoutine** to reset the *Cancel* routine to **NULL**.
5. If the driver owns power policy for more than one device, decrement its wait/wake reference count. If the count is nonzero, indicating that another device had previously sent a wait/wake IRP, request another wait/wake IRP (**PoRequestPowerIrp**) for its PDO.

   For example, a PCI device might have wait/wake enabled for both a modem and a Network Interface Card (NIC). If the NIC wakes the system (thus completing the IRP), the PCI FDO must send another wait/wake IRP to itself so that the modem will still be able to wake up.

Because the driver that requested the wait/wake IRP controls power policy for its device stack, it is responsible for returning its device to the working state when the IRP completes. Although lower drivers might already have physically applied power to the device, the policy owner must call **PoRequestPowerIrp** to send an **IRP_MN_SET_POWER** request for device power state D0. Only after all drivers in the device stack have handled this power-up IRP will the device be returned to the working state.

Canceling a Wait/Wake IRP

6/25/2019 • 3 minutes to read • Edit Online

Only the driver that sent a wait/wake IRP can cancel that IRP.

A driver might need to cancel a pending wait/wake IRP under the following circumstances:

- The driver receives a PnP **IRP_MN_STOP_DEVICE**, **IRP_MN_QUERY_REMOVE_DEVICE**, **IRP_MN_REMOVE_DEVICE**, or **IRP_MN_SURPRISE_REMOVAL** request for the device. The driver should reissue the wait/wake IRP (**PoRequestPowerIrp**) after the device is restarted.
- The system is going to sleep, but the device should not be enabled to wake the system.

  For example, the USB hub driver might send an **IRP_MN_WAIT_WAKE** request at device start-up in case it later puts one of its input devices into a sleep state. While the system is in the working state, a wake signal from the device returns the device to the working state (but has no effect on the system power state). When the system prepares to shut down, the USB hub driver cancels this IRP if the device should not be allowed to awaken the system.
- The system is entering a sleep state from which the device cannot awaken it. That is, it is entering a state less powered than the **SystemWake** value specified in its **DEVICE_CAPABILITIES** structure.
- The device is entering a power state from which it cannot respond to a wake-up signal. That is, it is entering a state less-powered than the **DeviceWake** value specified in its **DEVICE_CAPABILITIES** structure.

To cancel a wait/wake IRP, the driver that sent the IRP calls **IoCancelIrp**, passing the pointer to the IRP that was previously returned when the driver called **PoRequestPowerIrp**.

A driver must not cancel a wait/wake IRP that it did not send.

Cancel Routines for Wait/Wake IRPs

Many function and bus drivers should set *Cancel* routines for pending wait/wake IRPs; the following types of drivers must set such routines:

- Drivers that change device settings to enable or disable wake-up.
- Drivers that send **IRP_MN_WAIT_WAKE** requests to drivers of parent devices.

A *Cancel* routine allows a driver to disable wake-up for its device and to clean up any data related to the pending wait/wake IRP. Drivers that request wait/wake IRPs for parent devices can cancel those IRPs as well.

In its wait/wake *Cancel* routine, a driver should take the following steps:

1. Call **IoSetCancelRoutine** to reset the *Cancel* routine for the IRP to **NULL**.
2. Call **IoReleaseCancelSpinLock**, passing the **CancelIRQL** specified in the IRP to release the cancel spin lock for the IRP.
3. Reset any relevant fields in the device extension. For example, when a wait/wake IRP is pending, most drivers set a flag and keep a pointer to the IRP in the device extension.

   Note that it is possible for a driver to receive a wait/wake IRP while it is canceling another such IRP. The driver must check to see whether it already has an IRP under spin lock protection (or its equivalent). If so, the driver must carefully synchronize its handling to ensure that it cancels the correct IRP. For more information about using spin locks in *Cancel* routines, see Canceling IRPs.

4. Change any required device settings. For example, a modem driver would disable the device's wake setting.
5. Set **Irp->IoStatus.Status** to STATUS_CANCELLED.
6. Call **IoCompleteRequest** to complete the wait/wake IRP, specifying IO_NO_INCREMENT.
7. If the driver previously requested a related **IRP_MN_WAIT_WAKE** for a parent device, the driver should cancel that IRP from within its *Cancel* routine. The driver must release the cancel spin lock before it cancels the parent's IRP.

   For example, a driver that acts as a bus driver for a device and owns power policy driver for its parent should cancel a related wait/wake IRP that it earlier sent to its parent. Calling **IoCancelIrp** would invoke the parent's *Cancel* routine, and so on down the device stack.

Improving System Startup Performance

12/5/2018 • 2 minutes to read • Edit Online

One of the features that computer users most frequently request is fast startup times from power-off, standby, and hibernation states. To reduce the startup time, Windows uses a number of techniques, which include the following:

- Remove, from the list of startup operations, processes and services that can be deferred until after startup completes.
- Prefetch memory pages according to the pattern of requests to load these pages in previous system startups.
- Overlap device initialization with the disk I/O operations that are required to load the operating system.
- Enable device initializations to be performed in parallel instead of sequentially.

A kernel-mode driver should take the following steps to improve the performance of the startup process:

- When a computer starts up from a power-off state (cold startup), the device driver should do only what is required to initialize the device and defer all other device operations until startup is complete. Limit your driver's initialization code to the operations that are required to make the device ready to use.
- When a computer starts up from the standby or hibernation state (warm startup), a driver that must be initialized before startup completes should use high-priority worker threads and critical queue work items to offload any small tasks that it requires. Otherwise, the driver thread might be starved for processor time by unrelated threads, and startup will be delayed.
- During a warm startup from standby or hibernation, a driver's DPC routine, or initialization code that runs at DISPATCH_LEVEL, should avoid long execution times that block other drivers from running. For more information, see Sharing Processor Resources During Startup from a Low-Power State.
- During a warm startup from standby or hibernation, a functional device driver should complete an S0 set-power IRP immediately, and then request a D0 set-power IRP. If your driver promptly completes the S0 set-power IRP, the operating system can finish startup while your driver reinitializes the device as a background task. For more information, see Fast Startup from a Low-Power State.
- A device driver should not hold a spin lock for more than a brief time, especially during a cold startup from a power-off state. Otherwise, other device initializations cannot occur in parallel.

This section includes the following topics:

Sharing Processor Resources During Startup from a Low-Power State

6/25/2019 • 2 minutes to read • Edit Online

When a computer is started from a standby or hibernation state (warm startup), drivers should avoid using processor resources for longer than is necessary. Most importantly, deferred procedure call (DPC) routines and code that executes at IRQL >= DISPATCH_LEVEL should keep their execution times to a minimum. Drivers use DPC routines to help to initialize devices. Drivers might need to run initialization code at DISPATCH_LEVEL as part of a port-miniport interface contract.

While a DPC routine runs, other threads of lower priority are blocked from running on the same processor. In addition, other DPC routines that are queued and ready to run might be blocked until the current DPC is finished. To enable other threads to run expediently, a typical DPC routine should run for no more than 100 microseconds.

A DPC routine that runs for too long during system startup can delay the initialization of other devices. This delay makes the device initialization phase longer and delays startup completion by the operating system.

Use the following best practices to design your DPC routines:

- A single DPC routine should not execute for more than 100 microseconds.
- DPC routines that call the **KeStallExecutionProcessor** routine to delay execution must not specify delays of more than 100 microseconds.
- If a task requires longer than 100 microseconds and executes at DISPATCH_LEVEL, the DPC routine should end after 100 microseconds and schedule one or more DPC timer routines to complete the task at a later time.
- Use the performance analysis tools that are documented in the WDK to evaluate the execution times of DPC routines.

For more information about performance analysis tools, see Measuring System Resume Performance on Windows Vista.

Fast Startup from a Low-Power State

6/25/2019 • 3 minutes to read • Edit Online

To achieve a fast startup from a low-power state, a driver for a leaf-node device should handle an S0 power IRP (that is, an **IRP_MN_SET_POWER** IRP for the S0 system power state). Devices that are leaf nodes in the device hierarchy have no child devices. Because a leaf-node device has no dependencies on child devices, the functional driver for the device can reinitialize the device as a background task to avoid causing unnecessary delays to the operating system or to other drivers. In contrast, bus drivers have dependencies that require additional synchronization logic to coordinate power-on sequences with their child devices.

Use the following steps to achieve fast startup of a leaf-node device from a low-power state:

1. Set a completion routine for the S0 power IRP.
2. Send the S0 power IRP down the device stack.
3. Complete the S0 power IRP immediately instead of waiting until the D0 power IRP is completed. When the completion routine for the S0 power IRP runs, do the following:
    a. Request a D0 power IRP (that is, an **IRP_MN_SET_POWER** IRP for the D0 device power state).
    b. Return STATUS_SUCCESS to the completion routine for the S0 power IRP.
4. The driver should queue any I/O requests that it receives but defer handling any of these requests until it finishes processing the D0 power IRP.
5. When the completion routine for the D0 power IRP runs, initialize the device, but limit this routine to what is required to make the device ready to use.
6. After the previous steps are completed, your driver can begin to handle I/O requests, including any I/O requests that might already be queued.

Note The preceding steps do not apply to the handling of power IRPs for any power state other than PowerSystemWorking (S0). These steps specifically apply to the handling of power IRPs for transitions from a low-power state to the power-on (S0) state.

A system startup is complete after all devices have completed their S0 power IRPs. These devices are not required, at the completion of system startup, to have completed their D0 power IRPs or to be fully functioning. The kernel power manager has a limited set of IRP dispatch queues and must use these queues to notify all devices in the system of the return to the S0 state. Drivers that fail to quickly complete their S0 power IRPs prevent drivers for other devices from receiving their S0 power IRPs. Thus, poorly designed drivers impair overall system startup performance by causing driver operations that should be performed concurrently to be performed serially.

After a driver completes its S0 power IRP, it might receive I/O requests from applications that have opened handles to the device. Drivers must never fail these I/O requests because doing so might cause applications to stop responding and to produce time-out error messages. Instead, drivers must queue I/O requests until the device is ready to process them.

A bus driver can achieve a fast startup from a low-power state by using a technique similar to that just described for the driver of a leaf-node device. A bus driver must meet an additional requirement, which is to ensure that any requests from child devices to enter the D0 state are marked as pending and are not completed by the bus driver until the bus device has entered the D0 state.

For example, when the bus driver for a USB hub receives an S0 power IRP, the driver requests a D0 power IRP

and completes the S0 power IRP after receiving the requested D0 power IRP. However, after the S0 power IRP is completed, the hub's child devices are likely to start receiving their S0 power IRPs and requesting D0 power IRPs. The bus driver should prevent the child devices from entering D0 until the hub device enters D0. Therefore, the bus driver should mark all D0 power IRPs from child devices as pending and wait to complete these IRPs until the bus driver finishes handling the D0 power IRP for the hub and the hub device is fully initialized.

For more information about power IRPs, see the following topics:

Handling IRP_MN_SET_POWER for System Power States

Handling IRP_MN_SET_POWER for Device Power States

Device-Level Thermal Management

6/25/2019 • 2 minutes to read • Edit Online

Starting with Windows 8, Windows supports device-level thermal management for kernel-mode device drivers. Windows thermal management has these goals:

- Prevent devices in a hardware platform from overheating, which can cause them to operate incorrectly or unreliably.
- Avoid making user-accessible surfaces on a computer case too hot to comfortably touch or hold.

Similar to power management, thermal management must be implemented on a platform-wide basis by coordinating device-local thermal constraints in the context of global thermal conditions. By providing global coordination, the operating system can distribute cooling requirements across multiple devices in a way that minimizes interference with tasks that the user is performing. Thermal requirements can be balanced intelligently with other system requirements, such as power management and responsiveness to user actions.

In contrast, a device driver that tries to manage thermal levels for its device locally, in isolation from the other devices in the platform, is more likely to make poor decisions that result in inefficient power usage and an unresponsive user interface (UI).

To participate in global thermal management, a device driver implements a GUID_THERMAL_COOLING_INTERFACE driver interface. During system startup, a system-supplied driver, Acpi.sys, queries the device drivers in the system to determine which of them support this interface. A driver can receive an **IRP_MN_QUERY_INTERFACE** request for this interface any time after the *AddDevice* routine for the driver's device is called. In response to this request, the driver for a device that has thermal management capabilities can supply a pointer to a **THERMAL_COOLING_INTERFACE** structure. This structure contains pointers to a set of callback routines that are implemented by the driver. To manage thermal levels in the device, the operating system calls these routines directly.

The two principal routines in this interface are *ActiveCooling* and *PassiveCooling*. The driver's *ActiveCooling* routine engages or disengages active cooling in the device. For example, this routine might turn a fan on and off. The driver's *PassiveCooling* routine controls the degree to which the performance of the device must be throttled to maintain acceptable thermal levels. For example, this routine might be called to run the device at half speed to prevent it from overheating.

By default, before the first call to the *ActiveCooling* routine, active cooling is disengaged (for example, the fan is turned off). Before the first call to the *PassiveCooling* routine, the driver configures the device to run at full performance, with no cooling restrictions.

A driver can implement one or both of these routines, depending on the capabilities of the device hardware. For more information, see Passive and Active Cooling Modes.

Passive and Active Cooling Modes

12/5/2018 • 2 minutes to read • Edit Online

Starting with Windows 8, devices that have thermal management capabilities can expose these capabilities to the operating system through the GUID_THERMAL_COOLING_INTERFACE driver interface. The two principal driver-implemented callback routines in this interface are *PassiveCooling* and *ActiveCooling*. A driver that has passive-cooling capabilities implements the *PassiveCooling* routine. A driver that has active-cooling capabilities implements the *ActiveCooling* routine. In response to changes in computer usage or environmental conditions, the operating system calls one (or possibly both) of these routines to manage thermal levels dynamically in the hardware platform.

The Advanced Configuration and Power Interface (ACPI) enables the vendor for a hardware platform to partition the platform into regions called thermal zones. Sensor devices track the temperature in each thermal zone. When a thermal zone starts to overheat, the operating system can take actions to cool down the devices in the zone. These actions can be categorized as either passive cooling or active cooling.

To perform passive cooling, the operating system throttles one or more devices in the thermal zone to reduce the heat generated by these devices. Throttling might involve reducing the frequency of the clock that drives a device, lowering the voltage supplied to the device, or turning off a part of the device. As a rule, throttling limits device performance.

To perform active cooling, the operating system turns on a cooling device, such as a fan. Passive cooling decreases the power consumed by the devices in a thermal zone; active cooling increases power consumption.

In the design of a hardware platform, the decision to use passive cooling or active cooling is based on the physical characteristics of the hardware platform, the power source for the platform, and how the platform will be used.

Active cooling might be more straightforward to implement, but has several potential drawbacks. The addition of active cooling devices (for example, fans) might increase the cost and size of the hardware platform. The power required to run an active cooling device might reduce the time that a battery-powered platform can operate on a battery charge. Fan noise might be undesirable in some applications, and fans require ventilation.

Passive cooling is the only cooling mode available to many mobile devices. In particular, handheld computing platforms are likely to have closed cases and run on batteries. These platforms typically contain devices that can throttle performance to reduce heat generation. These devices include processors, graphics processing units (GPUs), battery chargers, and display backlights.

Handheld computing platforms typically use System on a Chip (SoC) chips that contain processors and GPUs, and the SoC hardware vendors supply the thermal management software for these devices. However, peripheral devices, such as battery chargers and display backlights, are external to SoC chips. The vendors for these devices must supply device drivers, and these drivers must provide any thermal management support that might be required for the devices. A relatively simple way for a device driver to support thermal management is to implement the GUID_THERMAL_COOLING_INTERFACE driver interface.

Global thermal management

12/5/2018 • 2 minutes to read • Edit Online

The GUID_THERMAL_COOLING_INTERFACE driver interface enables device drivers to participate in global thermal management across a variety of devices in the hardware platform. Drivers for devices that have thermal management capabilities implement the callback routines in this interface. The operating system calls these routines to dynamically manage thermal levels in the platform in response to changes in user activity and environmental conditions.

By preventing overheating, Windows thermal management keeps devices operating reliably and prevents user-accessible surfaces from becoming uncomfortably hot. Windows intelligently balances the thermal-level requirements of the devices in the platform to extend the time that the platform can operate on a battery charge, and to maintain the appearance of a computer that is always on and always connected.

For more information, see Device-Level Thermal Management.

Implementing WMI

6/25/2019 • 2 minutes to read • Edit Online

This section describes kernel-mode Windows Management Instrumentation (WMI) extensions to WDM. When you add these extensions to your kernel-mode driver, your driver becomes a WMI provider. A WMI provider makes measurement and instrumentation data available to WMI consumers, such as user-mode applications.

For more information about the user-mode WMI API, refer to Windows Management Instrumentation in the Windows SDK.

If you are implementing a KMDF-based driver, refer to Supporting WMI in Framework-Based Drivers.

This section includes the following information about kernel-mode WMI:

Introduction to WMI

12/5/2018 • 2 minutes to read • Edit Online

By making your driver a WMI provider, you can:

- Make custom data available to WMI consumers.
- Permit WMI consumers to configure a device through a standard interface rather than a custom control panel application.
- Notify WMI consumers of driver-defined events without requiring the consumer to poll or send IRPs.
- Reduce driver overhead by collecting and sending only requested data to a single destination.
- Annotate data and event blocks with descriptive driver-defined class names and optional descriptions that WMI clients can then enumerate and display to users.

WMI Architecture

12/5/2018 • 2 minutes to read • Edit Online

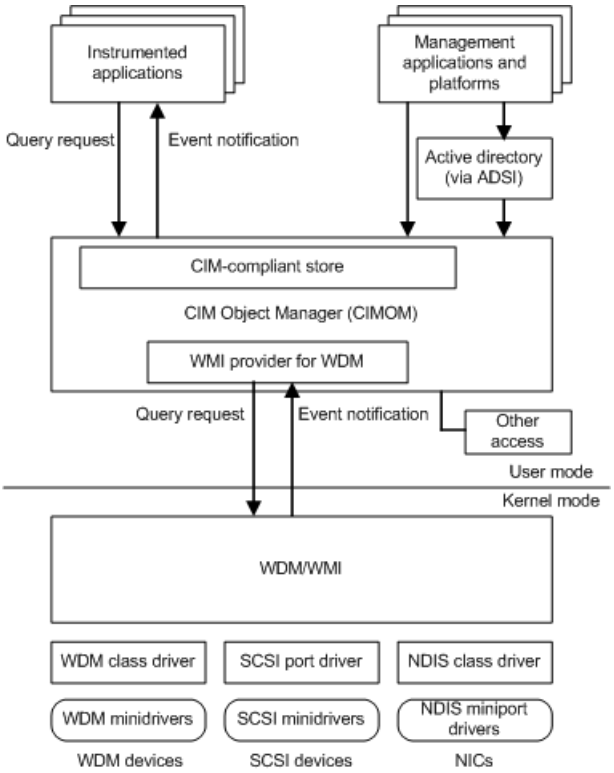
To support WMI, your driver registers as a WMI provider. A WMI provider is a Win32 dynamic-link library (DLL) that handles WMI requests and supplies WMI instrumentation data. See Registering as a WMI Data Provider to learn how a driver registers as a WMI provider.

After your driver is registered as a WMI provider, WMI consumers then request data or invoke methods exposed by providers.

Query requests travel from user-mode consumers down to the WMI kernel-mode service, which in turn sends IRP requests to your driver.

For instance, when a WMI client requests a given data block, the WMI kernel component sends a query request to the driver to retrieve or set data. The driver handles WMI requests as described in Handling WMI Requests.

The following figure shows this data flow:



WMI Requirements for WDM Drivers

6/25/2019 • 2 minutes to read • Edit Online

A driver that handles IRPs registers with WMI as a *data provider*. System-supplied storage port drivers, class drivers, and NDIS protocol drivers fall into this category. For information about registering as a WMI data provider, see Registering as a WMI Data Provider.

A driver that does not handle IRPs should simply forward WMI requests to the next-lower driver in the driver stack. The next-lower driver then registers with WMI and handles WMI requests on the first driver's behalf. For instance, SCSI miniport drivers and NDIS miniport drivers can register as WMI providers and supply WMI data to their corresponding class drivers.

A driver that supplies WMI data to a class or port driver must support the driver-type-specific WMI interfaces that are defined by the class or port driver. For example, a SCSI miniport driver must set **WmiDataProvider** to **TRUE** in the **PORT_CONFIGURATION_INFORMATION** structure and handle SRB_FUNCTION_WMI requests from the SCSI port driver.

Similarly, a connection-oriented NDIS miniport driver that defines custom data blocks must support OID_GEN_CO_SUPPORTED_GUIDS; otherwise, NDIS maps those OIDs and status indications returned from OID_GEN_SUPPORTED_LIST that are common and known to NDIS to GUIDs defined by NDIS.

The following sections describe how to support WMI in a driver that handles IRPs.

MOF Syntax for WMI Data and Event Blocks

12/5/2018 • 2 minutes to read • Edit Online

A driver's WMI schema describes its data blocks, which define the information that a driver can provide and the methods it can execute in response to WMI requests. A driver's schema also describes its event blocks, which are data blocks that the driver sends to WMI when a driver-determined event occurs for which a WMI client user has requested notification.

A driver writer defines a driver's schema in Managed Object Format (MOF). MOF is a compiled language created by the Desktop Management Task force (DMTF) and based on Interface Definition language (IDL). A driver's MOF file contains a MOF class definition for each data block and event block the driver exposes to WMI.

A MOF class definition for a WMI data block follows this syntax:

[*Required and optional class qualifiers***]**

class*ClassName* : *OptionalBaseClass* **{ [key, read] string InstanceName; [read] boolean Active; [** *Required and optional property qualifiers* **]** *datatype itemname1***; [** *Required and optional property qualifiers* **]** *datatype itemnameN***; };** The following topics describe the syntax elements shown above:

WMI Class Qualifiers

WMI Class Names and Base Classes

Required Items in WMI Classes

WMI Property Qualifiers

Driver-Defined WMI Data Items

WMI Class Examples

For a general discussion of MOF syntax as it pertains to WMI clients and other kinds of applications, see the Microsoft Windows SDK.

WMI Class Qualifiers

6/25/2019 • 2 minutes to read • Edit Online

The following table lists the required and optional MOF class qualifiers that can be used to describe a driver's WMI data blocks and event blocks.

An *embedded class*, which is a class used solely as a data item in another class and not exposed as a WMI data block, requires only the **WMI** and **Guid** qualifiers. The other qualifiers are irrelevant to embedded classes and are ignored. For more information about embedded classes, see Driver-defined WMI Data Items.

Dynamic and **Static** are standard MOF qualifiers. For information about other standard MOF qualifiers, see the Microsoft Windows SDK.

QUALIFIER	DESCRIPTION
Dynamic	Indicates that the data provider supplies instances of the data block at run time, rather than providing instances of static data in the MOF file. All data and event blocks that a driver registers with WMI must be defined with the **Dynamic** qualifier.
Static	Indicates that the data provider supplies instances of static data in the MOF file, rather than supplying instances of the data block at run time. A driver does not register static data blocks with WMI, because the static data resides in the WMI database. Classes marked as **Static** in the MOF file should not be registered by the driver's **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** handlers.
Provider("WMIProv")	(Required) Indicates that the provider of the class is a WMI provider.
WMI	(Required) Indicates that the class is a WMI class.
Description("*description-string***")**	(Optional) Specifies a description of the block for the locale specified by the **Locale** qualifier. If defined, WMI clients can display the description string to users. A driver writer can use **Description** to document a class.
Guid("*guid-string***")**	(Required) Specifies the GUID, in string format, that uniquely identifies the block to WMI. A driver writer should generate a GUID for each data block in the driver's MOF file, using either guidgen.exe or uuidgen.exe (which are included in the Windows SDK). A driver passes this value in GUID format to WMI when the driver registers its blocks. WMI then uses the GUID to look up the block's definition in the driver's MOF resource.

QUALIFIER	DESCRIPTION
Locale("MS*locale-identifier*")	(Optional) Specifies the language identifier and locale for the string specified by **Description**. For example, a *locale-identifier* of 0x409 specifies American English. A single MOF file can contain blocks with different locales, but typically all of the blocks in a MOF file have the same locale.
WmiExpense(*expense-value***)**	(Optional) Specifies the average number of CPU cycles needed to collect data for the data block. For example, a WMI client might check a data block's **WmiExpense** value to determine how often to query for its data. If **WmiExpense** is omitted, *expense-value* is assumed to be 0. **WmiExpense** is unrelated to registering a data block as expensive to collect.

WMI Class Names and Base Classes

3/5/2019 • 2 minutes to read • Edit Online

WMI class names are case-insensitive, must start with a letter, and cannot begin or end with an underscore. All remaining characters must be letters, digits, or underscores.

WMI client applications can access a driver's WMI class names and display them to users. Descriptive class names can help make classes more intuitive to use.

WMI class names must be unique within the WMI namespace. Consequently a driver's WMI class names cannot duplicate those defined by another driver.

To help prevent name collisions, a driver writer can define a driver-specific base class and derive all of the driver's WMI classes from that base class. The class name and base class name together are more likely to yield a unique name. For example, the following shows an abstract base class for a serial driver's data blocks:

```
// Serial driver's base class for data blocks
[abstract]
class MSSerial {
}

// Example class definition for a data block
[
    //Class qualifiers
]
class MSSerial_StandardSerialInformation : MSSerial
{
    //Data items
}
```

Device-specific custom data blocks should include the manufacturer, model, and type of driver or device in the base class name. For example:

```
[abstract]
class Adaptec1542 {
}

class Adaptec1542_Bandwidth : Adaptec1542 {
    //Data items
}

class Adaptec1542_Speed : Adaptec1542 {
    //Data items
}
```

WMI allows only one abstract base class in a given class hierarchy. Classes that define event blocks must derive from **WmiEvent**, which is an abstract base class, so the **abstract** qualifier cannot be used in a driver-defined base class for event blocks. Instead, derive a nonabstract base class from **WmiEvent**, then derive individual event classes from that base class. For example:

```
//Serial driver's base class for event blocks
class MSSerialEvent : WmiEvent
{
}

//Example class definition for an event block
[
    //Class qualifiers
]
class MSSerial_SendEvent : MSSerialEvent
{
    //Data items
}
```

For more information about defining base classes in MOF format, see the Microsoft Windows SDK.

Required Items in WMI Classes

12/5/2018 • 2 minutes to read • Edit Online

All class definitions except embedded classes must include the items **InstanceName** and **Active**, which must appear exactly as shown:

```
//WMI class definition
[
    //Class qualifiers
]
ClassName : BaseClassName
{
    [key, read]
     string InstanceName;
    [read]
     boolean Active;

    // Driver-defined data items
}
```

The **InstanceName** and **Active** items are required and used internally by WMI. The MOF class definitions of a driver's data and event blocks must include these items, but the driver must not set these items when responding to a query for the data block or sending an event, because they are not part of the driver's data block.

WMI Property Qualifiers

6/25/2019 • 3 minutes to read • Edit Online

The following table lists the required and optional MOF property qualifiers that can be used to define items in a WMI data or event block.

The following are standard MOF qualifiers: **key**, **read**, **write**, **ValueMap**, and **Values**. For more information about these and other standard MOF qualifiers, see MOF Data Types.

QUALIFIER	DESCRIPTION
key	Indicates that the data item is a key property that uniquely identifies each instance of the class. Only the InstanceName property can be declared a key.
read	Indicates that a WMI client can read the data item.
write	Indicates that a WMI client can set the data item.
BitMap	Specifies the bit positions of the corresponding string values that are specified in **BitValues**.
BitValues	Specifies a list of string values (flag names) that represent bits set in the data item. The bit position of a flag is defined by the corresponding position specified in **BitMap**.
DefineValues	Specifies an enumerated list that the WMI tool suite compiles into a corresponding list of #define statements.
DisplayInHex	Specifies that any WMI client that displays the property value should do so in hexadecimal.
DisplayName("*string***")**	Specifies a caption that a WMI client can use to display as the property name.
MaxLen(*uint***)**	For string properties, **MaxLen** specifies the maximum length of the string in characters. The *uint* value can be any 32-bit unsigned integer. If MaxLen is omitted, or *uint* is zero, then the length of the string is unlimited.
Values	Specifies a list of possible values for this data item. If the data item is an enumeration, **ValueMap** contains the index value that corresponds to the enumeration value specified in **Values**.

QUALIFIER	DESCRIPTION
ValueMap	Specifies the integer values of the corresponding string values in **Values**.
WmiDataId(*data-item-ID***)**	(Required) Identifies a data item within a data block. Data item IDs must be assigned to all items in a block except the required items **InstanceName** and **Active**. Data item IDs must be assigned in a contiguous series, starting with 1. An item's data ID determines the order in which the item appears in an instance of the data block; the order of items in the MOF class definition is irrelevant.
WmiMethodId(*method-item-ID***)**	Identifies a method within a data block.
WmiSizeIs("*data-item-name***")**	Specifies the name of another data item in this block that indicates the number of elements in the variable-length array at this data item. **WmiSizeIs** is valid only for data items that define arrays.
WmiScale(*scale-factor***)**	Specifies the scaling factor, as a power of 10, that the driver uses when returning the value of this data item. For example, if *scale-factor* is 5, the value returned by the driver is multiplied by 10^5. If **WmiScale** is omitted, *scale-factor* can be assumed to be 0.
WmiTimeStamp	Specifies that a 64-bit data item is a time stamp in units of 100 nanoseconds since 1/1/1601. **WmiTimeStamp** is valid only for 64-bit data items.
WmiComplexity(*level***)**	Specifies an integer value that expresses the user complexity level of the data item. WMI clients can use that value to distinguish between data items that should be available to novice users and data items that should be restricted to more advanced users. Zero is the minimum value, and higher values indicate higher user complexity. **WmiComplexity** defaults to zero if not specified.
WmiVolatility(*interval***)**	Specifies the interval, in milliseconds, between updates of this data item. For example, if a data item is updated once each second, *interval* would be 1000. A WMI client might check **WmiVolatility** to determine how often to query for a potentially new value. If **WmiVolatility** is omitted, *interval* is undefined.

QUALIFIER	DESCRIPTION
WmiEventTrigger(" *data-item-name***")**	Specifies the name of a data item in an event block that a WMI client can set to define the trigger value for the event. For example, if the event TooHot is qualified with **WmiEventTrigger**("TooHotTemperature"), a WMI client could set TooHotTemperature to instruct the driver to send the TooHot event when the device reached the user-specified value for TooHotTemperature. Typically a driver would define the trigger value. By exposing a **WmiEventTrigger** data item, the driver allows a client to control when a particular event is fired.
WmiEventRate("*data-item-name***")**	Specifies the name of a data item in an event block that a WMI client can set to control the frequency at which this event will be sent. For example, if the data item TooHot is qualified with **WmiEventRate("**SendEventRate**")**, a WMI client user could set SendEventRate to instruct the driver to send TooHot at the user-specified interval.

Driver-Defined WMI Data Items

6/25/2019 • 3 minutes to read • Edit Online

A data item in a class definition of WMI data or event block can be one of the following:

- A basic data type such as a string or an unsigned integer.
- An embedded class. An embedded class is used only as a data item in another class definition and is not exposed as a data block or event block.
- A fixed-length or variable-length array of a basic data type or embedded class.

When sending a data block to WMI, a driver must align the start of the block on an 8-byte boundary. All subsequent data items in the block must be aligned on the corresponding alignment for the data type. A **boolean** or **uint8** should be aligned on a 1-byte boundary. A **sint16**, **uint16**, or **string** item should be aligned on a 2-byte boundary, and so on. Arrays should be aligned based upon the base type of the array. An array of bytes should be aligned on a byte boundary, an array of uint64 should be aligned on an 8-byte boundary, and so on. An embedded class should be aligned based upon the natural alignment of the embedded class which is defined to be the largest element within the embedded class. For example, if an embedded class has a **uint64**, the class should be aligned on an 8-byte boundary. WMI data item alignment follows the same conventions as the **/Zp8** switch on the Microsoft C compiler.

A driver writer does not necessarily have to define data items in a block other than the required items **InstanceName** and **Active**. For example, an empty event block can serve as notification that an event occurred, without additional data. Or a data block might simply enumerate instance names in response to an **IRP_MN_QUERY_ALL_DATA** request.

The following table lists the MOF data types that can be used to define items in a WMI data or event block. For more information about MOF data types, see the Microsoft Windows SDK.

DATA TYPE	DATA FORMAT	ALIGNMENT (IN BYTES)
string	A USHORT specifying the string length in bytes, followed by the Unicode string data. The string data may optionally include a terminating 0 followed by padding. If so, the string length must include the terminating 0 and padding. Drivers can use the **MaxLen** qualifier to specify the maximum length in characters of the string. Drivers that specify a maximum string length can use a fixed size buffer to hold the string. If the string is strictly smaller than the size of the buffer, then the driver can pad the rest of the string with zeros.	2
boolean	A one-byte value where 0 is FALSE and any nonzero value is TRUE	1

DATA TYPE	DATA FORMAT	ALIGNMENT (IN BYTES)
sint8	Signed 8-bit integer	1
uint8	Unsigned 8-bit integer	1
sint16	Signed 16-bit integer	2
uint16	Unsigned 16-bit integer	2
sint32	Signed 32-bit integer	4
uint32	Unsigned 32-bit integer	4
sint64	Signed 64-bit integer	8
uint64	Unsigned 64-bit integer	8

DATA TYPE	DATA FORMAT	ALIGNMENT (IN BYTES)
datetime	A fixed-length 25-character Unicode string that specifies an absolute date or time interval. A **datetime** value has the following format: *yyyymmddhhmmss.mmmmmmsutc* where: *yyyy* is the 4-digit year *mm* is the 2-digit month *dd* is the 2-digit day of the month *hh* is the hour according to a 24-hour clock *mm* is the minute *ss* is the seconds *mmmmmm* is the number of microseconds *s* is a plus sign (+) or minus sign (-), indicating whether *utc* is a positive or negative offset from Universal Time Coordinates; or a colon (:), indicating that the **datetime** value is an interval. *utc* is the offset in minutes from Universal Time Coordinates. If *utc* is zero (000), the **datetime** value is an interval. Values must be zero-padded. Fields that are not significant can be filled with asterisks (*).	2

WMI Class Examples

12/19/2018 • 2 minutes to read • Edit Online

The following examples show class definitions from the schema of a serial port driver. Note that the **guid** values shown in these examples are placeholders. Each class definition must have a unique GUID generated by guidgen.exe or uuidgen.exe (which are included in the Microsoft Windows SDK).

```
// Standard class for reporting serial port information
// Class qualifiers
[WMI, guid("xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"),
Dynamic, Provider("WMIProv"),
WmiExpense(1),
Locale("MS\\0x409"),
Description("Description of class"]

//Class name
class Vendor_SerialInfo {

//Required items
    [key, read]
     string InstanceName;
    [read]
     boolean Active;

// Bytes sent on port
// Property qualifiers
    [read,
     WmiDataId(1),
     WmiScale(0),
     WmiComplexity(1),
     WmiVolatility(1000)]
     Description("Description of property")]
// Data item
     uint64 BytesSent;

// Bytes received on port
    [read,
     write,
     WmiDataId(2),
     WmiScale(0),
     WmiVolatility(1000)]
     uint64 BytesReceived;

// Who owns the port
    [read,
     WmiDataId(4),
     WmiScale(0),
     WmiVolatility(60000)]
    string Owner;

// Status bit array
    [read, write,
     WmiDataId(3),
     WmiScale(0)]
     byte Status[16];

//The number of items in the XmitBufferSize array
    [read,
     WmiDataId(5),
     WmiScale(0),
     WmiComplexity(1),
     WmiVolatility(1000)]
     uint32 XmitDescriptorCount;

//Array of XmitDescriptor classes
    [read,
     WmiDataId(6),
     WmiSizeIs("XmitDescriptorCount"),
     WmiScale(0),
     WmiComplexity(1),
     WmiVolatility(1000)]
    Vendor_XmitDescriptor XmitBufferSize[];
}
```

The following is the class definition for the embedded class shown in the previous example. Note that this class does not contain **InstanceName** or **Active** items.

```
// Example of an embedded class
[WMI, guid("xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"),
class Vendor_XmitDescriptor {
    [read, WmiDataId(1)] int32 DestinationIndex;
    [read, WmiDataId(2)] int32 DestinationTarget;
}
```

The following is a class definition for an event block. The class is derived from **WmiEvent**.

```
// Example of an event
[WMI, Dynamic, Provider("WMIProv"),
guid("{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}"),
locale("MS\\0x409"),
WmiExpense(1),
Description("Notify Toaster Arrival")]
class ToasterNotifyDeviceArrival : WMIEvent
{
    [key, read]
    string      InstanceName;

    [read]
    boolean           Active;

    [read,
     Description("Device Model Name"),
     WmiDataId(1)]    string     ModelName;
};
```

Designing WMI Data and Event Blocks

12/5/2018 • 2 minutes to read • Edit Online

For best performance and ease of use by WMI clients, a driver should support standard data blocks, and driver writers should follow certain guidelines in designing custom WMI data and event blocks. In particular, driver writers should be aware of performance tradeoffs in choosing static versus dynamic instance names for data blocks. The topics in this section discuss issues and guidelines for designing WMI data and event blocks.

Supporting Standard WMI Blocks

12/5/2018 • 2 minutes to read • Edit Online

Each driver should support any *standard blocks* defined for drivers of its type. Standard blocks provide WMI clients with consistent data for all devices of a given type, regardless of vendor.

To support a standard block, a driver:

- Registers the block with WMI, along with the other standard and custom blocks supported by the driver, as described in Registering as a WMI Data Provider.
- Handles all WMI requests that specify the driver's device object pointer at **Parameters.WMI.ProviderId** and the GUID of the standard block at **Parameters.WMI.DataPath**, as described in Handling WMI Requests.

MOF classes for standard blocks are published in system-supplied MOF files. A driver must not redefine a standard block in its own MOF file, because doing so would duplicate the block in the WMI database.

Currently, class definitions of standard blocks are published in the file wmicore.mof, which is included in the Windows Driver Kit (WDK).

Implementing Custom WMI Blocks

6/25/2019 • 3 minutes to read • Edit Online

A driver can implement *custom blocks* that expose device-specific instrumentation. For example, a driver for a disk drive that can report temperature might implement a custom event block that notifies WMI clients when the drive's temperature increases beyond a safe threshold.

To implement a custom block, a driver:

- Defines a class in its MOF file, compiles the MOF file into a resource, and includes the resource in the driver, as described in Publishing a WMI Schema.
- Registers the block with WMI along with the other standard and custom blocks supported by the driver, as described in Registering as a WMI Data Provider.
- Handles all WMI requests that specify the driver's device object pointer at **Parameters.WMI.ProviderId** and the GUID of the standard block at **Parameters.WMI.DataPath**, as described in Handling WMI Requests.

Drivers cannot control the order in which binary MOF files are loaded. The only guarantee is that wmicore.mof is loaded before any driver-specific MOF file. Therefore, custom WMI classes must only inherit from either classes in the same MOF file, or in wmicore.mof.

To improve performance and ease of use of custom WMI data blocks, consider the following guidelines:

- Put data items that are operationally grouped together in the same data block.

  For example, an i8042 port controller might maintain state information about both the keyboard and mouse ports. Rather than a single large data block containing all mouse and keyboard information, a driver might define one data block for the mouse port and another data block for the keyboard port.

- Put frequently used data items in separate data blocks, particularly if they would otherwise be grouped with items that are infrequently used.

  For example, a driver might expose CPU utilization in a data block with a single item, so a WMI client could track CPU utilization without incurring the overhead of retrieving additional data items in the block. A WMI client cannot query for a single data item, so to obtain one item it must query for an entire instance of a data block.

- Use event blocks to notify WMI clients of exceptional events, not as an alternative to error logging.

  Only a limited number of events can be queued at one time, and if the queue is full events will be lost. Also, the timing of delivery of events to WMI clients cannot be guaranteed.

- Limit event blocks to a maximum size of 1K bytes.

  Event items should be defined as small data types, because there is a registry-defined size limit (initially, 1K) for the entire **WNODE_EVENT_ITEM** structure that contains the generated event. For large notifications, a driver can send a **WNODE_EVENT_REFERENCE** structure that specifies a single instance of a data block, which WMI then queries to obtain the actual event. However, this increases the time lag between the occurrence of the event and the notification.

- Place fixed-size data items at the beginning of a data block, followed by any variable-size data items.

  For example, a data block that has three DWORD data items and one variable-length string should put the

three DWORDs first, followed by the string. Placing fixed-size data items at the beginning of a block permits WMI clients to extract them more easily.

- Consider which types of system users you'd like to access your driver's data blocks. The system provides a default security descriptor for all WMI class GUIDs. If necessary, you can provide alternate security descriptors within the device's INF file. For more information, see Creating Secure Device Installations.

WMI does not support versioning, so a driver writer must define a new MOF class and generate a new GUID to revise an existing custom block.

Defining WMI Instance Names

6/25/2019 • 2 minutes to read • Edit Online

An *instance* of a WMI block contains data supplied by a particular physical device or software component. Just as a block's GUID uniquely identifies the block, an instance's name uniquely identifies that instance of a block. WMI client applications use instance names to associate the information returned in a data block with the device or component that supplied the data. WMI uses instance names to determine which device a request should be sent to. It is strongly recommended that drivers use their PDO when defining instance names.

A driver can define instance names for a block in either of two ways:

- The driver passes a list of *static instance names* to WMI when it registers the block.

  After the block is registered, both the driver and WMI specify an instance name by its index into this list. Static instance names can be based on the *device instance ID* of a driver's PDO, or a driver-defined base name; or the driver can define a list of instance name strings. Static instance names persist until the driver explicitly changes them by reregistering the block.

- The driver generates *dynamic instance names* as instances are created.

  The driver indicates that it will generate dynamic instance names for a block when it registers the block. After the block is registered, both the driver and WMI pass dynamic instance names as strings in the buffer at **Parameters.WMI.Buffer**.

A driver should generate dynamic instance names only if the number of instances or instance names of a data block change frequently at runtime. For example, a driver might use process IDs or the IP addresses of TCP/IP connections as instance names. Such instance names should be dynamic; if they were static, the driver would incur considerable overhead because it would have to call **IoWMIRegistrationControl** to update the number and names of instances each time a change occurred.

In most cases, static instance names are preferable to dynamic instance names for the following reasons:

- Static instance names improve a driver's performance because the driver does not need to return instance name strings in response to WMI requests, as it must for dynamic instance names.

- WMI can detect static instance name collisions at registration and automatically modify the instance names if necessary, so that all instance names are unique for a given block no matter how many drivers register the block.

  WMI cannot detect instance name collisions for dynamic instance names, so the driver is responsible for generating unique names using **IoWMIAllocateInstanceIds**.

- A driver can use the WMI Library routines to handle IRPs for a block that uses static instance names, as long as the names are based on the driver's PDO or a driver-defined base name.

  A driver cannot use WMI Library routines to handle IRPs for a data block that uses dynamic instance names.

A driver indicates whether a block uses static or dynamic instance names, and the type of static instance names, by setting or clearing WMIREG_FLAG_*XXX* in the **WMIREGGUID** or **WMIGUIDREGINFO** structure it passes to WMI when it registers the block. For more information, see Registering as a WMI Data Provider.

Publishing a WMI Schema

12/5/2018 • 2 minutes to read • Edit Online

To publish a WMI schema, a driver writer first creates a text file in Managed Object Format (MOF) language that contains a class definition for each data block and event block in the schema, as described in MOF Syntax for WMI Data and Event Blocks.

A driver writer can then publish a driver's WMI schema in one of the following ways:

- Compile the MOF file and include it as a resource in the driver's binary image. For more information, see Compiling a Driver's MOF File.
- Include the compiled MOF file as a resource in a different file, such as a DLL, and add the registry value **MofImagePath** with a path to the file that contains the MOF under the driver's Service key. A schema published in this way can be updated without recompiling the driver. For more information, see Setting the MofImagePath Registry Value.
- Include binary data within the driver and return it when WMI requests it. A schema published in this way can be changed dynamically at runtime. For more information, see Implementing Dynamic MOF Data.

Compiling a Driver's MOF File

6/25/2019 • 2 minutes to read • Edit Online

To compile a MOF file that defines WMI data and event blocks, use the MOF compiler, called Mofcomp, that is included with the Microsoft Windows operating systems. Use the following syntax:

```
mofcomp -WMI -B:filename.bmf filename.mof
```

The following items appear in the preceding syntax:

-WMI
Validates all classes in *filename.mof* for use with WMI. If any class definition is invalid, Mofcomp deletes the output file *filename.bmf*. If **-WMI** is omitted, you should run Wmimofck on *filename.bmf* to validate the classes. A driver must either use the WMI switch or run Wmimofck to validate the MOF. Failure to do so can result in the MOF file not loading correctly into the WMI schema.

-B:*filename.bmf*
Requests that the compiler create a platform-independent binary version of the MOF file in *filename.bmf* without making any modifications to the CIMOM object repository.

filename.mof
Specifies the name of the input MOF file.

To learn more about how to use Mofcomp, open a Command Prompt window and type **mofcomp /?**.

For more information about Mofcomp, see MofComp and other topics in the Windows SDK.

To include the compiled MOF file as a resource in the driver's binary image, add the following line to the driver's resource script (RC) file:

MofResource MOFDATA *filename.bmf*

A driver specifies its MOF resource name in response to a registration request (an **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request with **Parameters.WMI.DataPath** set to WMIREGISTER):

- If the driver is using the WMI library routines to handle WMI IRPs, it specifies the MOF resource name in its *DpWmiQueryReginfo* routine.
- If the driver is handling WMI IRPs directly, it specifies the MOF resource name in the **WMIREGINFO** structure that the driver passes to WMI.

For more information about handling **IRP_MN_REGINFO** and **IRP_MN_REGINFO_EX** requests, see Registering as a WMI Data Provider.

For more information about handling WMI IRPs using WMI iibrary routines, see Handling WMI Requests.

For more information about defining and including resources in executable files, see the Microsoft Windows SDK.

Setting the MofImagePath Registry Value

6/25/2019 • 2 minutes to read • Edit Online

A driver's schema can be published by including a compiled MOF resource in a separate file, such as a DLL, and setting **MofImagePath** in the registry to the path of that file. A schema published in this way can be updated without recompiling the driver.

To publish a driver's schema in a separate file:

1. Compile the MOF file as described in Compiling a Driver's MOF File.
2. Include the compiled MOF file as a resource in a file such as a DLL.
3. Add the **MofImagePath** registry value under the driver's Services key. For example, the following shows the registry value for a driver named *DriverName*:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
    \DriverName
        MofImagePath    "\SystemRoot\System32\Drivers\DriverNameMof.dll"
```

You can set this key in the driver's INF file, as follows:

```
; This is the Services section for the driver
[Driver_service_install_section]
AddReg=Driver_AddReg

; This is the Services AddReg section declared above.
[Driver_AddReg]
HKR,,MofImagePath,,DriverMof.dll
```

See **INF DDInstall.Services Section** and **INF AddReg Directive** for details.

Implementing Dynamic MOF Data

6/25/2019 • 2 minutes to read • Edit Online

A driver's schema can be published dynamically by including binary MOF data in the driver's binary and returning selected schema information at runtime. To supply dynamic MOF data, a driver should follow these steps:

1. Compile the MOF file as described in Compiling a Driver's MOF File.
2. Use wmimofck.exe to create a .x file which will contain a hexadecimal dump of the .bmf file created by the MOF compiler.
3. Use **#include** to include the hex data created in step 2 with the driver's source.
4. Register as supporting MSWmi_MofData_GUID, which is a GUID defined in wmidata.h.
5. Return selected binary data to WMI in response to both the **IRP_MN_QUERY_ALL_DATA** or **IRP_MN_QUERY_SINGLE_INSTANCE** requests for MSWmi_MofData_GUID.

For more information about the wmimofck utility see Using wmimofck.exe.

Localizing MOF Files

12/5/2018 • 2 minutes to read • Edit Online

Since WMI property qualifiers are often displayable strings, Windows XP and later versions of the operating system provide a mechanism for localizing these qualifiers.

Creating the Localized MOF File

12/21/2018 • 2 minutes to read • Edit Online

On Windows XP and later versions of the operating system, drivers localize a WMI schema by making an *amended* version of each class. An amended version of a class updates property qualifiers that depend on the locale.

An amended version of a class has the same format as a class declaration, preceded by the **amendment** qualifier. The amended class declaration also includes a **locale(**0x*XXX***)** qualifier, where *XXX* specifies the locale identifier (LCID) for the locale.

The amended declaration includes the modified property declarations. Each localized property qualifier has the **:amended** modifier attached to it. For example, the localized version of **Description("***a description string***")** would be **Description("***localized description string***"):amended**.

Here is an example of a declaration of the basic class, followed by its amendment for American English.

```
[guid(xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxx)]
class MyClass
{
    [key] sint32 KeyProp;
    string Name;
    uint64 Timestamp;
}

[amendment, locale(0x409)
 Description("Localized version of MyClass for American English"):amended]

class MyClass
{
    [DisplayName("Key Property"):amended,
     Description("The description of KeyProp"):amended]
    sint32 KeyProp;

    [DisplayName("User Name"):amended,
     Description("The description of Name"):amended]
    string Name;
}
```

Only the properties that have been modified need to be included in the amended class. The class and property names cannot be localized. Only property qualifiers can be localized.

Localized classes are organized in child namespaces of the namespace containing the original class. Classes for a given locale are found in the MS_*XXX* child namespace, where *XXX* represents the hexadecimal LCID for that locale. For example, drivers are in the \root\wmi namespace by default. An amended class, localized for American English, is found in the \root\wmi\MS_409 namespace.

For more information about WMI localization, see the WMI international support website.

Building and Deploying the Localized MOF File

12/5/2018 • 2 minutes to read • Edit Online

International versions of Windows XP and later versions of the operating system come in two flavors — single-language (localized) versions, and Multilanguage User Interface (MUI) versions. An MUI version of Windows can support several languages simultaneously.

Drivers that are deployed on a localized version of Windows should contain a MOF resource that contains the language-neutral version of all the classes, as well as the localized language amendment and the American English language amendment.

On an MUI version of Windows, the driver image itself should contain the language-neutral and American English versions of the WMI classes. For each additional language supported, a resource-only image can be placed in the %windir%\system32\drivers\MUI*langid* directory, where *langid* is the LCID of the for the locale.

Optionally, the driver image itself can contain every language supported.

If a language is not supported by one of these mechanisms, the English language version is used.

Building Distinct MOF Files For Each Language

Driver writers can use one master MOF text file to contain the basic class, and all of its amendments.

You can use the MOF compiler to generate a file containing the language-neutral classes as well as a file to contain the amended classes for a particular language.

```
mofcomp -amendment:namespace [ -MOF:mof] [ -MFL:mfl] masterfile
```

The *namespace* parameter is of the form MS_*XXX*, where *XXX* is the LCID for the locale to be generated. The mof file created contains the language-neutral classes, and the mfl file created contains the amended classes.

When building your driver on NT-based operating systems, you can merge the two files by using the copy command. On Windows 98/Me, the copy command does not correctly append Unicode files, but the following command can be used.

```
wmimofck localizedfile -ymof -zmfl
```

You can combine any number of languages into a single text file.

The localized file can then be compiled into a binary file by the same method as for the MOF files that have not been localized:

```
mofcomp -B:binaryfile localizedfile
```

For a single-language version of Windows, drivers attach the binary MOF as a resource to the driver image. See Compiling a Driver's MOF File for details.

On an MUI system, the driver image itself must be built for American English. For each additional language, install each localized binary MOF file as a resource in a separate image file in the appropriate %windir%\system32\drivers\MUI*langid* directory, where *langid* is the hexadecimal LCID for the locale (for example, 409 for American English). The file name must be either *drivername.sys* or *drivername.sys*.mui, where *drivername.sys* is the name of the driver binary.

Building One MOF File for All Supported Languages

If the driver image will contain every supported language, there is a simpler way to build a MOF file supporting every language. By using **#pragma** directives in the MOF text file, drivers can also combine all of the amended classes in one binary. Since each localization exists in a distinct namespace, they can safely be combined in a single binary.

When writing the combined MOF text file, driver writers must precede each amended class declaration with a **#pragma** directive as follows

```
#pragma namespace ("namespace")
```

where `namespace` is the correct namespace for the declaration. For example, the amended class declaration for American English must be preceded with a declaration of the form:

```
#pragma namespace ("\\\\.\\root\\wmi\\ms_409")
```

For example, you declare a class and its amendments as follows.

```
#pragma namespace ("\\\\.\\root\\wmi)

[guid(xxxxxxxx-xxxx-xxxx-xxxxxxxxxxxx)]
class MyClass
{
}

#pragma namespace(\\\\.\\root\\wmi\\ms_409)
[amendment, locale(0x407)]
class MyClass
{
}
```

Using this approach, building the binary MOF file is identical to the nonlocalized approach. See Compiling a Driver's MOF File for details.

Registering as a WMI Data Provider

6/25/2019 • 2 minutes to read • Edit Online

A driver that supports WMI must register as a WMI data provider to make its data and event blocks available to WMI clients. A driver typically registers with WMI when starting its device, after the device has been initialized to the point that the driver can handle WMI IRPs. During the registration process, the driver passes WMI a pointer to its device object and information about the data and event blocks it supports.

A driver registers with WMI in two phases:

1. The driver calls **IoWMIRegistrationControl** with the action WMIREG_ACTION_REGISTER and a pointer to the device object passed to the driver's *AddDevice* routine.

2. The driver handles the **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request that WMI sends in response to the driver's **IoWMIRegistrationControl** call. The **Parameters.WMI.DataPath** member of the IRP is set to WMIREGISTER and **Parameters.WMI.ProviderId** is set to the driver's device object pointer. The driver supplies WMI with registration information about its data and event blocks, either by using the WMI Library as described in Using the WMI Library to Register Blocks, or by handling the **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** requests as described in Handling IRP_MN_REGINFO and IRP_MN_REGINFO_EX to Register Blocks.

Using the WMI Library to Register Blocks

6/25/2019 • 2 minutes to read • Edit Online

A driver can use the WMI Library to handle **IRP_MN_REGINFO** and **IRP_MN_REGINFO_EX** requests if it is registering blocks that do not use dynamic instance names, or that use static instance names based on a PDO or driver-defined base name string. In this case, the driver:

1. Calls **WmiSystemControl** with a pointer to the driver's device object, a pointer to a **WMILIB_CONTEXT** structure, and a pointer to the IRP

   The **WMILIB_CONTEXT** structure indicates the number of blocks to register (**GuidCount**) and points to a list of **WMIGUIDREGINFO** structures (**GuidList**) that specify the GUID, the number of instances, and registration flags that pertain to the corresponding block. It also defines entry points for the driver's required and optional *DpWmiXxx* callback routines.

2. When WMI calls the driver's *DpWmiQueryReginfo* routine, the driver specifies the driver's registry path, its MOF resource name, registration flags that pertain to all of its blocks, and information that WMI uses to name instances of the driver's data blocks, which could be either a pointer to the physical device object passed to the driver's *AddDevice* routine or a string on which to base static instance names.

A driver must initialize entry points for its *DpWmiXxx* callback routines in the **WMILIB_CONTEXT** structure before calling **WmiSystemControl**, but can postpone initialization of **GuidCount** and **GuidList** in the **WMILIB_CONTEXT** structure until WMI calls the driver's *DpWmiQueryReginfo* routine.

Handling IRP_MN_REGINFO and IRP_MN_REGINFO_EX to Register Blocks

6/25/2019 • 2 minutes to read • Edit Online

On Windows 98 and Windows 2000, the system sends the **IRP_MN_REGINFO** request to a driver to allow a driver to register its WMI classes. On Windows XP and later, the system sends the **IRP_MN_REGINFO_EX** request instead. Most drivers can handle these requests by using **WmiSystemControl** to provide a callback routine. See Using the WMI Library to Register Blocks for details.

A driver must handle **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** requests to register blocks that use dynamic instance names or that use a list of driver-defined static instance names; it cannot call **WmiSystemControl** to register such blocks. A driver can optionally handle this request to register blocks that use static instance names based on the PDO or a driver-defined base name string.

In this case, the driver:

1. Fills in a **WMIREGINFO** structure at **Parameters.WMI.Buffer** that specifies:
   - The number of bytes of all registration data supplied by the driver, including data supplied on behalf of another driver.
   - The driver's registry path.
   - The name of the driver's MOF resource.
   - The number of blocks to register.
   - An array of **WMIREGGUID** structures, one for each block.
2. For each block, the driver fills in a **WMIREGGUID** structure that specifies:
   - The GUID that represents the block.
   - Flags that provide information about instance names and other characteristics of the block, such as whether the block is expensive to collect. For more information, see WMI Registration Flags.

   If the block is being registered with static instance names, the driver sets one of the following members to specify static instance name data for the block:

   - If the driver sets **Flags** with WMIREG_FLAG_INSTANCE_LIST, it sets **InstanceNameList** to an offset to a list of static instance name strings. WMI specifies instances in subsequent requests by index into this list.
   - If the driver sets **Flags** with WMIREG_FLAG_INSTANCE_BASENAME, it sets **BaseNameOffset** to an offset to a base name string. WMI uses this string to generate static instance names for the block.
   - If the driver sets **Flags** with WMIREG_FLAG_INSTANCE_PDO, it sets **Pdo** to the PDO passed to the driver's *AddDevice* routine. WMI uses the device instance path of the PDO to generate static instance names for the block. When handling an **IRP_MN_REGINFO_EX** request, drivers must call the **ObReferenceObject** routine on the physical device object passed in **Pdo**. (The system will automatically call **ObDereferenceObject** to dereference the object; the driver must not do so.)

   The driver writes instance name strings or a base name string at the offset indicated by **InstanceNameList** or **BaseName**, respectively.

3. If the driver is registering blocks on behalf of another driver (as a class driver might on behalf of a miniclass driver), the driver fills in another **WMIREGINFO** structure and list of **WMIREGGUID** structures with registration information for the other driver's blocks, and sets **NextWmiRegInfo** in the first **WMIREGINFO** to the offset in bytes from the beginning of the first **WMIREGINFO** to the beginning of the second **WMIREGINFO** structure.

WMI Registration Flags

6/25/2019 • 2 minutes to read • Edit Online

A driver indicates whether a block uses static or dynamic instance names and specifies other characteristics of the block by setting flags in the **WMIGUIDREGINFO** or **WMIREGGUID** structure that it passes to WMI to register the block.

A driver indicates that a block uses static instance names by setting one of the following flags:

- WMIREG_FLAG_INSTANCE_LIST indicates that the driver provides all instance names in a static list.

  A driver can set this flag only if it registers blocks by handling the **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** requests, not by calling **WmiSystemControl**. The driver writes the instance name strings at the byte offset indicated by **InstanceNameList** in the block's **WMIREGGUID** structure.

- WMIREG_FLAG_INSTANCE_BASENAME instructs WMI to generate static instance names from a driver-defined base name string.

  A driver that handles an **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request writes the base name string at the offset indicated by **BaseNameOffset** in the block's **WMIREGGUID** structure.

  A driver that calls **WmiSystemControl** specifies the base name string in the *InstanceName* parameter of its **DpWmiQueryReginfo** routine.

- WMIREG_FLAG_INSTANCE_PDO instructs WMI to generate static instance names from the device instance ID of the driver's PDO.

  A driver that handles an **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request writes a pointer to the PDO at the **Pdo** member of the block's **WMIREGGUID** structure. If the request is **IRP_MN_REGINFO_EX**, the driver must increase the reference count on each PDO passed by calling the **ObReferenceObject** routine. (The system will dereference each PDO later.)

  A driver that calls **WmiSystemControl** writes a pointer to the PDO in the *Pdo* parameter of its *DpWmiQueryReginfo* routine.

To indicate that a block uses dynamic instance names, the driver must not set any of the following flags: WMIREG_FLAG_INSTANCE_LIST, WMIREG_FLAG_INSTANCE_PDO, or WMIREG_FLAG_INSTANCE_BASENAME.

A driver indicates that a data block is expensive to collect by setting WMIREG_FLAG_EXPENSIVE. This instructs WMI to send an **IRP_MN_ENABLE_COLLECTION** request the first time a WMI client opens the data block and an **IRP_MN_DISABLE_COLLECTION** request when the last WMI client closes the block. The driver need not collect data for such a block until it receives an **IRP_MN_ENABLE_COLLECTION** request.

A driver indicates an event block by setting WMIREG_FLAG_EVENT_ONLY_GUID. This indicates that the block can be enabled or disabled as an event only, and cannot be queried or set.

A driver instructs WMI to remove a previously registered block by setting WMIREG_FLAG_REMOVE_GUID. This flag is valid only in response to a request to update registration information (**IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** with WMIUPDATE). For more information, see Updating WMI Registration Information.

Updating WMI Registration Information

6/25/2019 • 2 minutes to read • Edit Online

After its initial registration with WMI, a driver changes its registration information by calling **IoWMIRegistrationControl** with one of the following actions:

- WMIREG_ACTION_REREGISTER to replace all registration information previously supplied by the driver with new information.

  In response, WMI sends either an **IRP_MN_REGINFO** request or an **IRP_MN_REGINFO_EX** request to the driver, with **Parameters.WMI.DataPath** set to WMIREGISTER. (On Windows 98 and Windows 2000, the system sends the **IRP_MN_REGINFO** request. On Windows XP and later, the system sends the **IRP_MN_REGINFO_EX** request.)

  The driver supplies WMI with new registration information for all blocks it supports, as described in Using the WMI Library to Register Blocks and Handling IRP_MN_REGINFO and IRP_MN_REGINFO_EX to Register Blocks.

- WMIREG_ACTION_UPDATE_GUIDS to add or remove support for blocks or to change the static instance names of registered blocks.

  In response, WMI sends an **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request to the driver, with **Parameters.Wmi.DataPath** set to WMIUPDATE.

  The driver supplies WMI with an updated registration information in which:

  - The driver sets WMIREG_FLAG_REMOVE_GUID to remove support for that block.
  - The driver clears WMIREG_FLAG_REMOVE_GUID to add a new block or update an existing block.
  - The driver sets or clears WMIREG_FLAG_INSTANCE_*XXX* and supplies any necessary instance name information to change a block's static instance names or change it to use dynamic instance names.

- WMIREG_ACTION_DEREGISTER to instruct WMI that the driver will no longer provide WMI information.

  WMI does not send an **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request in response to this call, because it requires no further information from the driver. A driver typically deregisters its blocks in response to an **IRP_MN_REMOVE_DEVICE** request. Note that the deregister call will block until all WMI IRPs to the device have been completed. If a driver queues WMI IRPs, it must cancel them before calling **IoWMIRegistrationControl** to deregister.

Handling WMI Requests

6/25/2019 • 2 minutes to read • Edit Online

All drivers must set a dispatch table entry point for a *DispatchSystemControl* routine to handle WMI requests. If a driver registers as a WMI data provider, it must handle all WMI requests. Otherwise, the driver must forward all WMI requests to the next lower driver.

All WMI IRPs have the major code **IRP_MJ_SYSTEM_CONTROL** and a one of the following minor codes:

- **IRP_MN_REGINFO**, **IRP_MN_REGINFO_EX**—Queries or updates a driver's registration information after the driver has called **IoWMIRegistrationControl**.
- **IRP_MN_QUERY_ALL_DATA**, **IRP_MN_QUERY_SINGLE_INSTANCE**—Queries for all instances or a single instance of a given data block.
- **IRP_MN_CHANGE_SINGLE_ITEM**, **IRP_MN_CHANGE_SINGLE_INSTANCE**—Requests the driver to change a single item or multiple items in an instance of a data block.
- **IRP_MN_ENABLE_COLLECTION**, **IRP_MN_DISABLE_COLLECTION**—Requests the driver to start accumulating data for a block that the driver registered as expensive to collect, or to stop accumulating data for such a block.
- **IRP_MN_ENABLE_EVENTS**, **IRP_MN_DISABLE_EVENTS**—Requests the driver to start sending notification of a given event if the event occurs while it is enabled, or to stop sending notification of such an event.
- **IRP_MN_EXECUTE_METHOD**—Requests the driver to execute a method associated with a data block.

The WMI kernel-mode component sends WMI IRPs any time following a driver's successful registration as a WMI data provider, typically when a user-mode data consumer has requested WMI information for a driver's device. If a driver registers as a WMI data provider by calling **IoWMIRegistrationControl**, it must handle each subsequent WMI request in one of the following ways:

- Call the kernel-mode WMI library routine **WmiSystemControl** of a PDO. For more information, see Calling WmiSystemControl to Handle WMI IRPs.
- In its *DispatchSystemControl* routine, process and complete any such request tagged with the pointer to its device object that the driver passed in its call to **IoWMIRegistrationControl**, and forward other **IRP_MJ_SYSTEM_CONTROL** requests to the next lower driver. For more information, see Processing WMI IRPs in a DispatchSystemControl Routine.

For a list of the WMI minor IRPs, see WMI Minor IRPs.

WMI Minor IRPs

6/25/2019 • 2 minutes to read • Edit Online

This section describes the Windows Management Instrumentation IRPs that are part of the WMI extensions to WDM. All WMI IRPs use the major code **IRP_MJ_SYSTEM_CONTROL** and a minor code that indicates the specific WMI request. The WMI kernel-mode component can send WMI IRPs any time following a driver's successful registration as a supplier of WMI data. WMI IRPs typically get sent when a user-mode data consumer has requested WMI data.

All drivers must set a dispatch table entry point for a *DispatchSystemControl* routine to handle WMI requests.

If a driver registers as a WMI data provider by calling **IoWMIRegistrationControl**, it must handle WMI IRPs using one of the techniques described in Handling WMI Requests.

Drivers that do not register as WMI data providers must forward all WMI requests to the next-lower driver.

This section describes the following system-defined WMI minor function codes:

IRP_MN_CHANGE_SINGLE_INSTANCE

IRP_MN_CHANGE_SINGLE_ITEM

IRP_MN_DISABLE_COLLECTION

IRP_MN_DISABLE_EVENTS

IRP_MN_ENABLE_COLLECTION

IRP_MN_ENABLE_EVENTS

IRP_MN_EXECUTE_METHOD

IRP_MN_QUERY_ALL_DATA

IRP_MN_QUERY_SINGLE_INSTANCE

IRP_MN_REGINFO

IRP_MN_REGINFO_EX

If the driver receives an IRP containing any other IRP minor function code, it should forward the IRP to the next-lower driver.

IRP_MN_CHANGE_SINGLE_INSTANCE

6/25/2019 • 3 minutes to read • Edit Online

All drivers that support WMI must handle this IRP. A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver calls **WmiSystemControl** to handle an **IRP_MN_CHANGE_SINGLE_INSTANCE** request, WMI in turn calls that driver's *DpWmiSetDataBlock* routine.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

WMI sends this IRP to change all data items in a single instance of a data block.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is found in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath points to a GUID that identifies the data block associated with the instance to be changed.

Parameters.WMI.BufferSize indicates the size of the nonpaged buffer at **Parameters.WMI.Buffer**.

Parameters.WMI.Buffer points to a **WNODE_SINGLE_INSTANCE** structure that identifies the instance and specifies new data values.

Output Parameters

None.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_WMI_INSTANCE_NOT_FOUND

STATUS_WMI_GUID_NOT_FOUND

STATUS_WMI_READ_ONLY

STATUS_WMI_SET_FAILURE

On success, the driver sets **Irp->IoStatus.Information** to zero.

Operation

If a driver handles WMI IRPs by calling **WmiSystemControl**, that routine calls the driver's *DpWmiSetDataBlock* routine, or returns STATUS_WMI_READ_ONLY if the driver does not define the routine.

If a driver handles an **IRP_MN_CHANGE_SINGLE_INSTANCE** request itself, it does so only if the device object pointer at **Parameters.WMI.ProviderId** matches the pointer passed by the driver in its call to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next-lower driver.

If the driver handles the request, it must first check the GUID at **Parameters.WMI.DataPath** to determine whether it identifies a data block supported by the driver. If not, the driver must fail the IRP and return STATUS_WMI_GUID_NOT_FOUND.

If the driver supports the data block, it must check the received **WNODE_SINGLE_INSTANCE** structure at **Parameters.WMI.Buffer** for the instance name, as follows:

- If WNODE_FLAG_STATIC_INSTANCE_NAMES is set in **WnodeHeader.Flags**, the driver uses **InstanceIndex** as an index into the driver's list of static instance names for that block. WMI obtains the index from registration data provided by the driver when it registered the block.
- If WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags,** the driver uses the offset at **OffsetInstanceName** to locate the instance name string in the input **WNODE_SINGLE_INSTANCE**. **OffsetInstanceName** is the offset in bytes from the beginning of the structure to a USHORT-sized length of the instance name string in bytes (not characters), including the terminating null if present, followed by the instance name string in Unicode.

The driver is responsible for validating all input values. Specifically, the driver must do the following if it handles the IRP request itself:

- For static names, verify that the **InstanceIndex** member of the **WNODE_SINGLE_INSTANCE** structure is within the range of instance indexes supported by the driver for the data block.
- For dynamic names, verify that the instance name string identifies a data block instance supported by the driver.
- Verify that the **DataBlockOffset** and **SizeDataBlock** members of the **WNODE_SINGLE_INSTANCE** structure describe a valid-sized data block, including any padding that exists between data items, and that the contents of the buffer are valid for the data block.
- Verify that the specified data block is one for which the driver allows caller-initiated modifications. In other words, the driver should not allow modifications to data blocks that you intended to be read-only.

Do not assume the thread context is that of the initiating user-mode application — a higher-level driver might have changed it.

If the driver cannot locate the specified instance, it must fail the IRP and return STATUS_WMI_INSTANCE_NOT_FOUND. If the instance has a dynamic instance name, this status indicates that the driver does not support the instance. WMI can therefore continue to query other data providers, and return an appropriate error to the data consumer if another provider finds the instance but cannot handle the request for some other reason.

If the driver locates the instance and can handle the request, it sets the writable data items in the instance to the values in the **WNODE_SINGLE_INSTANCE** structure, leaving any read-only items unchanged. If the entire data block is read-only, the driver should fail the IRP and return STATUS_WMI_READ_ONLY.

If the instance is valid but the driver cannot handle the request, it can return any appropriate error status.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiSetDataBlock

IoWMIRegistrationControl

WMILIB_CONTEXT

WmiSystemControl

WNODE_SINGLE_INSTANCE

IRP_MN_CHANGE_SINGLE_ITEM

6/25/2019 • 3 minutes to read • Edit Online

All drivers that support WMI must handle this IRP. A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver calls **WmiSystemControl** to handle an **IRP_MN_CHANGE_SINGLE_ITEM** request, WMI in turn calls that driver's *DpWmiSetDataItem* routine.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

WMI sends this IRP to change a single data item in a single instance of a data block.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath points to a GUID that identifies the data block to be set.

Parameters.WMI.BufferSize indicates the size of the nonpaged buffer at **Parameters.WMI.Buffer**.

Parameters.WMI.Buffer, points to a **WNODE_SINGLE_ITEM** structure that identifies the instance of the data block, the ID of the item to set, and a new data value.

Output Parameters

None.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_WMI_INSTANCE_NOT_FOUND

STATUS_WMI_ITEMID_NOT_FOUND

STATUS_WMI_GUID_NOT_FOUND

STATUS_WMI_READ_ONLY

STATUS_WMI_SET_FAILURE

On success, a driver sets **Irp->IoStatus.Information** to zero.

Operation

If a driver handles WMI IRPs by calling **WmiSystemControl**, that routine calls the driver's *DpWmiSetDataItem* routine, or returns STATUS_WMI_READ_ONLY if the driver does not define the routine.

If a driver handles **IRP_MN_CHANGE_SINGLE_ITEM** requests itself, it should do so only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next-lower driver.

Do not implement support for **IRP_MN_CHANGE_SINGLE_ITEM** unless you are sure that a system-supplied user-mode component requires this capability.

Before handling a request, the driver must determine whether **Parameters.WMI.DataPath** points to a GUID that the driver supports. If it does not, the driver must fail the IRP and return STATUS_WMI_GUID_NOT_FOUND.

If the driver supports the data block, it must check the input **WNODE_SINGLE_ITEM** structure that **Parameters.WMI.Buffer** points to for the instance name, as follows:

- If WNODE_FLAG_STATIC_INSTANCE_NAMES is set in **WnodeHeader.Flags**, the driver uses **InstanceIndex** as an index into the driver's list of static instance names for that block. WMI obtains the index from registration data provided by the driver when it registered the block.
- If WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags,** the driver uses the offset at **OffsetInstanceName** to locate the instance name string in the input **WNODE_SINGLE_ITEM** structure. **OffsetInstanceName** is the offset in bytes from the beginning of the structure to a USHORT-sized length of the instance name string in bytes (not characters). This length includes the NULL terminator if present, followed by the instance name string in Unicode.

The driver is responsible for validating all input values. Specifically, the driver must do the following if it handles the IRP request itself:

- For static names, verify that the **InstanceIndex** member of the WNODE_SINGLE_ITEM structure is within the range of instance indexes supported by the driver for the data block.
- For dynamic names, verify that the instance name string identifies a data block instance supported by the driver.
- Verify that the **ItemId** member of the **WNODE_SINGLE_ITEM** structure is within the range of item identifiers supported by the driver for the data block.
- Verify that the **DataBlockOffset** and **SizeDataItem** members of the **WNODE_SINGLE_ITEM** structure describe a valid-sized data block, and that the contents of the buffer are valid for the data item.
- Verify that the specified data item is one for which the driver allows caller-initiated modifications. In other words, the driver should not allow modifications to data items that you intended to be read-only.

Do not assume the thread context is that of the initiating user-mode application—a higher-level driver might have changed it.

If the driver cannot locate the specified instance, it must fail the IRP and return STATUS_WMI_INSTANCE_NOT_FOUND. For an instance with a dynamic instance name, this status indicates that the driver does not support the instance. WMI can therefore continue to query other data providers, and return an appropriate error to the data consumer if another provider finds the instance but cannot handle the request for some other reason.

If the driver locates the instance and can handle the request, it sets the data item in the instance to the value in the **WNODE_SINGLE_ITEM**. If the data item is read-only, the driver leaves the item unchanged, fails the IRP, and returns STATUS_WMI_READ_ONLY.

If the instance is valid but the driver cannot handle the request, it can return any appropriate error status.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiSetDataItem

IoWMIRegistrationControl

WMILIB_CONTEXT

WmiSystemControl

WNODE_SINGLE_ITEM

IRP_MN_DISABLE_COLLECTION

6/25/2019 • 2 minutes to read • Edit Online

Any WMI driver that registers one or more of its data blocks as expensive to collect must handle this IRP. A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver calls **WmiSystemControl** to handle an **IRP_MN_DISABLE_COLLECTION** request, WMI in turn calls that driver's *DpWmiFunctionControl* routine.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

WMI sends this IRP to request the driver to stop accumulating data for a data block that the driver registered as expensive to collect and for which data collection has been enabled.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath points to a GUID that identifies the data block for which data accumulation should be stopped.

Output Parameters

None.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_WMI_GUID_NOT_FOUND

STATUS_INVALID_DEVICE_REQUEST

On success, a driver sets **Irp->IoStatus.Information** to zero.

Operation

A driver registers a data block as expensive to collect by setting WMIREG_FLAG_EXPENSIVE in the **Flags** member of the **WMIREGGUID** or **WMIGUIDREGINFO** structure that the driver passes to WMI when it registers or updates the data block. A driver need not accumulate data for such a block until it receives an explicit request to enable collection.

If a driver handles WMI IRPs by calling **WmiSystemControl**, that routine calls the driver's

DpWmiFunctionControl routine, or returns STATUS_SUCCESS if the driver does not define the routine.

If a driver handles an **IRP_MN_DISABLE_COLLECTION** request itself, it should do so only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next-lower driver.

Before handling the request, the driver must determine whether **Parameters.WMI.DataPath** points to a GUID that the driver supports. If not, the driver must fail the IRP and return STATUS_WMI_GUID_NOT_FOUND. If the data block is valid but was not registered with WMIREG_FLAG_EXPENSIVE, the driver can return STATUS_SUCCESS and take no further action.

It is unnecessary for the driver to check whether data collection is already disabled because WMI sends a single disable request for the data block when the last data consumer disables collection for that block. WMI will not send another disable request without an intervening request to enable.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiFunctionControl

IoWMIRegistrationControl

IRP_MN_ENABLE_COLLECTION

WMILIB_CONTEXT

WMIREGGUID

WMIGUIDREGINFO

WmiSystemControl

IRP_MN_DISABLE_EVENTS

6/25/2019 • 2 minutes to read • Edit Online

Any WMI driver that registers one or more event blocks must handle this IRP. A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver calls **WmiSystemControl** to handle an **IRP_MN_DISABLE_EVENTS** request, WMI in turn calls that driver's *DpWmiFunctionControl* routine.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

WMI sends this IRP to inform the driver that a data consumer has requested no further notification of an event.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath points to a GUID that identifies the event block to disable.

Output Parameters

None.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_WMI_GUID_NOT_FOUND

STATUS_INVALID_DEVICE_REQUEST

On success, a driver sets **Irp->IoStatus.Information** to zero.

Operation

A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver handles WMI IRPs by calling **WmiSystemControl**, that routine calls the driver's *DpWmiFunctionControl* routine, or returns STATUS_SUCCESS if the driver does not define the routine.

If a driver handles an **IRP_MN_DISABLE_EVENTS** request itself, it should do so only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next-lower driver.

Before handling a request, the driver must determine whether **Parameters.WMI.DataPath** points to a GUID the driver supports. If not, the driver must fail the IRP and return STATUS_WMI_GUID_NOT_FOUND.

If the driver supports the event block, it disables the event for all instances of that block.

It is unnecessary for the driver to check whether events are already disabled for the event block because WMI sends a single disable request for that event block when the last data consumer disables the event. WMI will not send another disable request without an intervening request to enable.

For details about defining event blocks, see Designing WMI Data and Event Blocks.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiFunctionControl

IoWMIRegistrationControl

IRP_MN_ENABLE_EVENTS

WMILIB_CONTEXT

WmiSystemControl

IRP_MN_ENABLE_COLLECTION

6/25/2019 • 2 minutes to read • Edit Online

Any WMI driver that registers one or more of its data blocks as potentially time-consuming, or *expensive*, to collect must handle this IRP. A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver calls **WmiSystemControl** to handle an **IRP_MN_ENABLE_COLLECTION** request, WMI in turn calls that driver's *DpWmiFunctionControl* routine.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

WMI sends this IRP to request the driver to start accumulating data for a data block that the driver registered as expensive to collect.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath points to a GUID that identifies the data block for which data is accumulated.

Output Parameters

None.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_WMI_GUID_NOT_FOUND

STATUS_INVALID_DEVICE_REQUEST

On success, a driver sets **Irp->IoStatus.Information** to zero.

Operation

A driver registers a data block as expensive to collect by setting WMIREG_FLAG_EXPENSIVE in the **Flags** member of the **WMIREGGUID** or **WMIGUIDREGINFO** structure. The driver passes these structures to WMI when it registers or updates the data block. A driver need not accumulate data for such a block until it receives an explicit request to start data collection.

A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver handles WMI IRPs by calling **WmiSystemControl**, that routine calls the driver's *DpWmiFunctionControl* routine, or returns STATUS_SUCCESS if the driver does not define the routine.

If a driver handles an **IRP_MN_ENABLE_COLLECTION** request itself, it should do so only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next-lower driver.

Before handling a request, the driver should make sure that **Parameters.WMI.DataPath** points to a GUID that the driver supports. If it does not, the driver should fail the IRP and return STATUS_WMI_GUID_NOT_FOUND. If the data block is valid but was not registered with WMIREG_FLAG_EXPENSIVE, the driver can return STATUS_SUCCESS and take no further action.

If the block is valid and was registered with WMIREG_FLAG_EXPENSIVE, the driver enables data collection for all instances of that data block.

It is unnecessary for the driver to check whether data collection is already enabled for the data block. WMI sends only a single request to enable a data block after the first data consumer enables the block. WMI will not send another request to enable without an intervening disable request.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiFunctionControl

IoWMIRegistrationControl

IRP_MN_DISABLE_COLLECTION

WMILIB_CONTEXT

WMIREGGUID

WmiSystemControl

IRP_MN_ENABLE_EVENTS

6/25/2019 • 2 minutes to read • Edit Online

Any WMI driver that registers one or more event blocks must handle this IRP. A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver calls **WmiSystemControl** to handle an **IRP_MN_ENABLE_EVENTS** request, WMI in turn calls that driver's *DpWmiFunctionControl* routine.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

WMI sends this IRP to inform the driver that a data consumer has requested notification of an event.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath points to a GUID that identifies the event block to enable.

Parameters.WMI.BufferSize indicates the size of the nonpaged buffer at **Parameters.WMI.Buffer**, which must be greater than or equal to the **sizeof**(**WNODE_HEADER**). A driver that does not register trace blocks (WMIREG_FLAG_TRACED_GUID) can ignore this parameter.

Parameters.WMI.Buffer points to a **WNODE_HEADER** that indicates whether the event should be traced (WMI_FLAGS_TRACED_GUID) and provides a handle to the system logger. A driver that does not register trace blocks (WMIREG_FLAG_TRACED_GUID) can ignore this parameter.

Output Parameters

None.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_WMI_GUID_NOT_FOUND

STATUS_INVALID_DEVICE_REQUEST

On success, a driver sets **Irp->IoStatus.Information** to zero.

Operation

A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in

Handling WMI Requests.

If a driver handles WMI IRPs by calling **WmiSystemControl**, that routine calls the driver's *DpWmiFunctionControl* routine, or returns STATUS_SUCCESS if the driver does not define the routine.

If a driver handles an **IRP_MN_ENABLE_EVENTS** request itself, it should do so only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next-lower driver.

Before the driver handles the request, it should determine whether **Parameters.WMI.DataPath** points to a GUID that the driver supports. If not, the driver must fail the IRP and return STATUS_WMI_GUID_NOT_FOUND.

If the driver supports the event block, it enables the event for all instances of that data block.

It is unnecessary for the driver to check whether events are already enabled for the event block because WMI sends a single request to enable for the event block when the first data consumer enables the event. WMI will not send another request to enable without an intervening disable request.

A driver that registers trace blocks (WMIREG_FLAG_TRACED_GUID) must also determine whether to send the event to WMI or to the system logger for tracing. If tracing is requested, **Parameters.WMI.Buffer** points to a **WNODE_HEADER** structure in which **Flags** is set with WNODE_FLAG_TRACED_GUID and **HistoricalContext** contains a handle to the logger.

For details about defining event blocks, sending events, and tracing, see Windows Management Instrumentation.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiFunctionControl

IoWMIRegistrationControl

IRP_MN_DISABLE_EVENTS

WMILIB_CONTEXT

WmiSystemControl

WNODE_EVENT_ITEM

WNODE_HEADER

IRP_MN_EXECUTE_METHOD

6/25/2019 • 5 minutes to read • Edit Online

All drivers that support methods within data blocks must handle this IRP. A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver calls **WmiSystemControl** to handle an **IRP_MN_EXECUTE_METHOD** request, WMI in turn calls that driver's *DpWmiExecuteMethod* routine.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

WMI sends this IRP to execute a method associated with a data block.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

WMI will send an **IRP_MN_QUERY_SINGLE_INSTANCE** prior to sending an **IRP_MN_EXECUTE_METHOD**. If a driver supports **IRP_MN_EXECUTE_METHOD** it must have a **IRP_MN_QUERY_SINGLE_INSTANCE** handler for the same data block whose method is being executed.

Input Parameters

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath points to a GUID that identifies the data block associated with the method to execute.

Parameters.WMI.BufferSize indicates the size of the nonpaged buffer at **Parameters.WMI.Buffer** which must be >= **sizeof**(**WNODE_METHOD_ITEM**) plus the size of any output data for the method.

Parameters.WMI.Buffer points to a **WNODE_METHOD_ITEM** structure in which **MethodID** indicates the identifier of the method to execute and **DataBlockOffset** indicates the offset in bytes from the beginning of the structure to the first byte of input data, if any. **Parameters.WMI.Buffer->SizeDataBlock** indicates the size in bytes of the input **WNODE_METHOD_ITEM** including input data, or zero if there is no input.

Output Parameters

If the driver handles WMI IRPs by calling **WmiSystemControl**, WMI fills in the **WNODE_METHOD_ITEM** with data returned by the driver's *DpWmiExecuteMethod* routine.

Otherwise, the driver fills in the **WNODE_METHOD_ITEM** structure that **Parameters.WMI.Buffer** points to as follows:

- Updates **WnodeHeader.BufferSize** with the size of the output **WNODE_METHOD_ITEM**, including any output data.
- Updates **SizeDataBlock** with the size of the output data, or zero if there is no output data.
- Checks **Parameters.WMI.Buffersize** to determine whether the buffer is large enough to receive the output **WNODE_METHOD_ITEM** including any output data. If the buffer is not large enough, the driver fills in the needed size in a **WNODE_TOO_SMALL** structure pointed to by **Parameters.WMI.Buffer**. If

the buffer is smaller than **sizeof**(**WNODE_TOO_SMALL**), the driver fails the IRP and returns STATUS_BUFFER_TOO_SMALL.

- Writes output data, if any, over input data starting at **DataBlockOffset**. The driver must not change the input value of **DataBlockOffset**.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_BUFFER_TOO_SMALL

STATUS_WMI_GUID_NOT_FOUND

STATUS_WMI_INSTANCE_NOT_FOUND

STATUS_WMI_ITEMID_NOT_FOUND

On success, a driver sets **Irp->IoStatus.Information** to the number of bytes written to the buffer at **Parameters.WMI.Buffer**.

Operation

A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver handles WMI IRPs by calling **WmiSystemControl**, that routine calls the driver's *DpWmiExecuteMethod* routine, or returns STATUS_INVALID_DEVICE_REQUEST if the driver does not define the routine.

If a driver handles an **IRP_MN_EXECUTE_METHOD** request itself, it must do so only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next-lower driver.

The driver is responsible for validating all input values. Specifically, the driver must do the following if it handles the IRP request itself:

- For static names, verify that the **InstanceIndex** member of the **WNODE_METHOD_ITEM** structure is within the range of instance indexes supported by the driver for the data block.
- For dynamic names, verify that the instance name string identifies a data block instance supported by the driver.
- Verify that the **MethodId** member of the **WNODE_METHOD_ITEM** structure is within the range of method identifiers supported by the driver for the data block, and that the caller is allowed to execute the method.
- Verify that the **DataBlockOffset** and **SizeDataBlock** members of the **WNODE_METHOD_ITEM** structure describe a buffer that is large enough to contain the specified method's parameters, and that the parameters are valid for the method.
- Verify that **Parameters.WMI.Buffersize** specifies a buffer that is large enough to receive the **WNODE_METHOD_ITEM** structure after it has been updated with output data.

Do not assume the thread context is that of the initiating user-mode application — a higher-level driver might

have changed it.

Before handling the request, the driver must determine whether **Parameters.WMI.DataPath** points to a GUID supported by the driver. If not, the driver must fail the IRP and return STATUS_WMI_GUID_NOT_FOUND.

If the driver supports the data block, it checks the input **WNODE_METHOD_ITEM** at **Parameters.WMI.Buffer** for the instance name, as follows:

- If WNODE_FLAG_STATIC_INSTANCE_NAMES is set in **WnodeHeader.Flags**, the driver uses **InstanceIndex** as an index into the driver's list of static instance names for that block. WMI obtains the index from registration data that was provided by the driver when it registered the block.
- If WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags,** the driver uses the offset at **OffsetInstanceName** to locate the instance name string in the input **WNODE_METHOD_ITEM**. **OffsetInstanceName** is the offset in bytes from the beginning of the structure to a USHORT which is the length of the instance name string in bytes (not characters), including the terminating null if present, followed by the instance name string in Unicode.

If the driver cannot locate the specified instance, it must fail the IRP and return STATUS_WMI_INSTANCE_NOT_FOUND. For an instance with a dynamic instance name, this status indicates that the driver does not support the instance. WMI can therefore continue to query other data providers, and return an appropriate error to the data consumer if another provider finds the instance but cannot handle the request for some other reason.

The driver then checks the method ID in the input **WNODE_METHOD_ITEM** to determine whether it is a valid method for that data block. If not, the driver fails the IRP and returns STATUS_WMI_ITEMID_NOT_FOUND.

If the method generates output, the driver should check the size of the output buffer in **Parameters.WMI.BufferSize** before performing any operation that might have side effects or that should not be performed twice. For example, if a method returns the values of a group of counters and then resets the counters, the driver should check the buffer size (and fail the IRP if the buffer is too small) before resetting the counters. This ensures that WMI can safely resend the request with a larger buffer.

If the instance and method ID are valid and the buffer is adequate in size, the driver executes the method. If **SizeDataBlock** in the input **WNODE_METHOD_ITEM** is nonzero, the driver uses the data starting at **DataBlockOffset** as input for the method.

If the method generates output, the driver writes the output data to the buffer starting at **DataBlockOffset** and sets **SizeDataBlock** in the output **WNODE_METHOD_ITEM** to the number of bytes of output data. If the method has no output data, the driver sets **SizeDataBlock** to zero. The driver must not change the input value of **DataBlockOffset**.

If the instance is valid but the driver cannot handle the request, it can return any appropriate error status.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiExecuteMethod

IoWMIRegistrationControl

WMILIB_CONTEXT

WmiSystemControl

WNODE_METHOD_ITEM

IRP_MN_QUERY_ALL_DATA

6/25/2019 • 3 minutes to read • Edit Online

All drivers that support WMI must handle this IRP. A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver calls **WmiSystemControl** to handle an **IRP_MN_QUERY_ALL_DATA** request, WMI in turn calls that driver's *DpWmiQueryDataBlock* routine.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

WMI sends this IRP to query for all instances of a given data block.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

Parameters.WMI.ProviderId in the driver's I/O stack location in the IRP points to the device object of the driver that should respond to the request.

Parameters.WMI.DataPath points to a GUID that identifies the data block.

Parameters.WMI.BufferSize indicates the maximum size of the nonpaged buffer at **Parameters.WMI.Buffer**, which receives output data from the request. The buffer size must be greater than or equal to **sizeof**(**WNODE_ALL_DATA**) plus the sizes of instance names and data for all instances to be returned.

Output Parameters

If the driver handles WMI IRPs by calling **WmiSystemControl**, WMI fills in a **WNODE_ALL_DATA** by calling the driver's *DpWmiQueryDataBlock* routine once for each block registered by the driver.

Otherwise, the driver fills in a **WNODE_ALL_DATA** structure at **Parameters.WMI.Buffer** as follows:

- Sets **WnodeHeader.BufferSize** to the number of bytes of the entire **WNODE_ALL_DATA** to be returned, sets **WnodeHeader.Timestamp** to the value returned by **KeQuerySystemTime**, and sets **WnodeHeader.Flags** as appropriate for the data to be returned.
- Sets **InstanceCount** to the number of instances to be returned.
- If the block uses dynamic instance names, sets **OffsetInstanceNameOffsets** to the offset in bytes from the beginning of the **WNODE_ALL_DATA** to where an array of ULONG offsets begins. Each element in this array is the offset from the **WNODE_ALL_DATA** to where each dynamic instance name is stored. Each dynamic instance name is stored as a counted Unicode string where the count is a USHORT followed by the Unicode string. The count does not include any terminating null character that may be part of the Unicode string. If the Unicode string does include a terminating null character, this null character must still fit within the size established in **WNodeHeader.BufferSize**.
- If all instances are the same size:
  - Sets WNODE_FLAG_FIXED_INSTANCE_SIZE in **WnodeHeader.Flags** and sets **FixedInstanceSize** to that size, in bytes.

- Writes instance data starting at **DataBlockOffset**, with padding so that each instance is aligned to an 8-byte boundary. For example, if **FixedInstanceSize** is 6, the driver adds 2 bytes of padding between instances.

- If instances vary in size:
  - Clears WNODE_FLAG_FIXED_INSTANCE_SIZE in **WnodeHeader.Flags** and writes an array of **InstanceCount OFFSETINSTANCEDATAANDLENGTH** structures starting at **OffsetInstanceDataAndLength**. Each **OFFSETINSTANCEDATAANDLENGTH** structure specifies the offset in bytes from the beginning of the **WNODE_ALL_DATA** structure to the beginning of the data for each instance, and the length of the data. **DataBlockOffset** is not used.
  - Writes instance data following the last element of the **OffsetInstanceDataAndLength** array, plus padding so that each instance is aligned to an 8-byte boundary.

If the buffer at **Parameters.WMI.Buffer** is too small to receive all of the data, a driver fills in the needed size in a **WNODE_TOO_SMALL** structure at **Parameters.WMI.Buffer**. If the buffer is smaller than **sizeof**(**WNODE_TOO_SMALL**), the driver fails the IRP and returns STATUS_BUFFER_TOO_SMALL.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_BUFFER_TOO_SMALL

STATUS_WMI_GUID_NOT_FOUND

On success, a driver sets **Irp->IoStatus.Information** to the number of bytes written to the buffer at **Parameters.WMI.Buffer**.

Operation

A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver handles WMI IRPs by calling **WmiSystemControl**, that routine calls the driver's *DpWmiQueryDataBlock* routine.

If a driver handles an **IRP_MN_QUERY_ALL_DATA** request, it should do so only if **Parameters.WMI.ProviderId** points to the same device object that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next-lower driver.

Before handling the request, the driver must determine whether **Parameters.WMI.DataPath** points to a GUID that the driver supports. If not, the driver must fail the IRP and return STATUS_WMI_GUID_NOT_FOUND.

If the driver supports the data block, it must do the following:

- Verify that **Parameters.WMI.BufferSize** specifies a buffer that is large enough to receive all the data that the driver will return.
- Fill in a **WNODE_ALL_DATA** structure at **Parameters.WMI.Buffer** with data for all instances of that data block.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiQueryDataBlock

IoWMIRegistrationControl

KeQuerySystemTime

WMILIB_CONTEXT

WmiSystemControl

WNODE_ALL_DATA

IRP_MN_QUERY_SINGLE_INSTANCE

6/25/2019 • 4 minutes to read • Edit Online

All drivers that support WMI must handle this IRP. A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver calls **WmiSystemControl** to handle an **IRP_MN_QUERY_SINGLE_INSTANCE** request, WMI in turn calls that driver's *DpWmiQueryDataBlock* routine.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

WMI sends this IRP to query for a single instance of a given data block.

WMI sends an **IRP_MN_QUERY_SINGLE_INSTANCE** prior to sending an **IRP_MN_EXECUTE_METHOD**. If a driver supports **IRP_MN_EXECUTE_METHOD**, it must have an **IRP_MN_QUERY_SINGLE_INSTANCE** handler for the same data block whose method is being executed.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in an arbitrary thread context.

Input Parameters

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath points to a GUID that identifies the data block to query.

Parameters.WMI.BufferSize indicates the maximum size of the nonpaged buffer at **Parameters.WMI.Buffer**, which points to a **WNODE_SINGLE_INSTANCE** structure that identifies the instance to query.

Output Parameters

If the driver handles WMI IRPs by calling **WmiSystemControl**, WMI fills in a **WNODE_SINGLE_INSTANCE** structure with data provided by the driver's *DpWmiQueryDataBlock* routine.

Otherwise, the driver fills in the **WNODE_SINGLE_INSTANCE** structure at **Parameters.WMI.Buffer** as follows:

- Updates **WnodeHeader.BufferSize** with the size, in bytes, of the output **WNODE_SINGLE_INSTANCE** structure, including instance data. This value should include the length of the instance name (padded such that the instance data begins on a quad word boundary), even if the class being queried registered static instance names and the driver writer is not explicitly supplying the name when servicing this IRP.
- Sets **SizeDataBlock** to the size, in bytes, of the instance data. If static instance names are in use, this value should not include the size of the instance name.
- Writes the instance data to **Parameters.WMI.Buffer** starting at **DataBlockOffset**. The driver must not change the input value of **DataBlockOffset**.

If the buffer at **Parameters.WMI.Buffer** is too small to receive all of the data, the driver fills in the needed size in a **WNODE_TOO_SMALL** structure at **Parameters.WMI.Buffer**. If the buffer is smaller than **sizeof**(**WNODE_TOO_SMALL**), the driver fails the IRP and returns STATUS_BUFFER_TOO_SMALL.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_BUFFER_TOO_SMALL

STATUS_WMI_GUID_NOT_FOUND

STATUS_WMI_INSTANCE_NOT_FOUND

On success, a driver sets **Irp->IoStatus.Information** to the value entered into **WnodeHeader.BufferSize**. This value includes the length of the static instance name.

Operation

A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver handles WMI IRPs by calling **WmiSystemControl**, **WmiSystemControl** calls the driver's *DpWmiQueryDataBlock* routine.

If a driver handles an **IRP_MN_QUERY_SINGLE_INSTANCE** request itself, it should do so only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed in its call to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next lower driver in the device stack.

Before handling the request, the driver must determine whether **Parameters.WMI.DataPath** points to a GUID that the driver supports. If not, the driver must fail the IRP and return STATUS_WMI_GUID_NOT_FOUND.

The driver is responsible for validating all input values. Specifically, the driver must do the following if it handles the IRP request itself:

- For static names, verify that the **InstanceIndex** member of the **WNODE_SINGLE_INSTANCE** structure is within the range of instance indexes supported by the driver for the data block.
- For dynamic names, verify that the instance name string identifies a data block instance supported by the driver.
- Verify that **Parameters.WMI.BufferSize** specifies a buffer that is large enough to receive all the data that the driver will return.

If the driver supports the data block, it checks the input **WNODE_SINGLE_INSTANCE** at **Parameters.WMI.Buffer** for the instance name, as follows:

- If WNODE_FLAG_STATIC_INSTANCE_NAMES is set in **WnodeHeader.Flags**, the driver uses **InstanceIndex** as an index into the driver's list of static instance names for that block. WMI obtains the index from registration data provided by the driver when it registered the block.
- If WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags**, the driver uses the offset at **OffsetInstanceName** to locate the instance name string in the input **WNODE_SINGLE_INSTANCE**. **OffsetInstanceName** is the offset, in bytes, from the beginning of the structure to a USHORT, which is the length of the instance name string in bytes (not characters), including the terminating null if present, followed by the instance name string in Unicode.

If the driver cannot locate the specified instance, it must fail the IRP and return

STATUS_WMI_INSTANCE_NOT_FOUND. For an instance with a dynamic instance name, this status indicates that the driver does not support the instance. WMI can therefore continue to query other data providers, and return an appropriate error to the data consumer if another provider finds the instance but cannot handle the request for some other reason.

If the driver locates the instance and can handle the request, it fills in the **WNODE_SINGLE_INSTANCE** structure at **Parameters.WMI.Buffer** with data for the instance.

If the instance is valid but the driver cannot handle the request, it can return any appropriate error status.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiQueryDataBlock

IoWMIRegistrationControl

WMILIB_CONTEXT

WmiSystemControl

WNODE_SINGLE_INSTANCE

IRP_MN_REGINFO

6/25/2019 • 5 minutes to read • Edit Online

Drivers that support WMI on Microsoft Windows 98 and Microsoft Windows 2000 must handle this IRP. (Drivers that support Windows XP as well must also handle the **IRP_MN_REGINFO_EX** IRP.) A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

On Windows 98 and Windows 2000, WMI sends this IRP to query or update a driver's registration information after the driver has called **IoWMIRegistrationControl**. On Windows XP and later, WMI sends the **IRP_MN_REGINFO_EX** request instead.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in the context of a system thread.

Input Parameters

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath is set to **WMIREGISTER** to query registration information or **WMIUPDATE** to update it.

Parameters.WMI.BufferSize indicates the maximum size of the nonpaged buffer at **Parameters.WMI.Buffer**. The size must be greater than or equal to the total of (**sizeof**(**WMIREGINFO**) + (*GuidCount* * **sizeof**(**WMIREGGUID**)), where *GuidCount* is the number of data blocks and event blocks being registered by the driver, plus space for static instance names, if any.

Output Parameters

If the driver handles WMI IRPs by calling **WmiSystemControl**, WMI gets registration information for a driver's data blocks by calling its *DpWmiQueryReginfo* routine.

Otherwise, the driver fills in a **WMIREGINFO** structure at **Parameters.WMI.Buffer** as follows:

- Sets **BufferSize** to the size in bytes of the **WMIREGINFO** structure plus associated registration data.
- If the driver handles WMI requests on behalf of another driver, sets **NextWmiRegInfo** to the offset in bytes from the beginning of this **WMIREGINFO** to the beginning of another **WMIREGINFO** structure that contains registration information from the other driver.
- Sets **RegistryPath** to the registry path that was passed to the driver's **DriverEntry** routine.
- If **Parameters.WMI.Datapath** is set to **WMIREGISTER**, sets **MofResourceName** to the offset from the beginning of this **WMIREGINFO** to a counted Unicode string that contains the name of the driver's MOF resource in its image file.
- Sets **GuidCount** to the number of data blocks and event blocks to register or update.
- Writes an array of **WMIREGGUID** structures, one for each data block or event block exposed by the driver, at **WmiRegGuid**.

The driver fills in each **WMIREGGUID** structure as follows:

- Sets **Guid** to the GUID that identifies the block.
- Sets **Flags** to provide information about instance names and other characteristics of the block. For example, if a block is being registered with static instance names, the driver sets **Flags** with the appropriate WMIREG_FLAG_INSTANCE_*XXX* flag.

If the block is being registered with static instance names, the driver:

- Sets **InstanceCount** to the number of instances.
- Sets one of the following members to an offset in bytes to static instance name data for the block:
  - If the driver sets **Flags** with WMIREG_FLAG_INSTANCE_LIST, it sets **InstanceNameList** to an offset to a list of static instance name strings. WMI specifies instances in subsequent requests by index into this list.
  - If the driver sets **Flags** with WMIREG_FLAG_INSTANCE_BASENAME, it sets **BaseNameOffset** to an offset to a base name string. WMI uses this string to generate static instance names for the block.
  - If the driver sets **Flags** with WMIREG_FLAG_INSTANCE_PDO, it sets **Pdo** to an offset to a pointer to the PDO passed to the driver's *AddDevice* routine. WMI uses the device instance path of the PDO to generate static instance names for the block.
- Writes the instance name strings, the base name string, or a pointer to the PDO at the offset indicated by **InstanceNameList**, **BaseName**, or **Pdo**, respectively.

If the driver handles WMI registration on behalf of another driver (such as a miniclass or miniport driver), it fills in another WMIREGINFO structure with the other driver's registration information and writes it at **NextWmiRegInfo** in the previous structure.

If the buffer at **Parameters.WMI.Buffer** is too small to receive all of the data, a driver writes the needed size in bytes as a ULONG to **Parameters.WMI.Buffer** and fails the IRP and returns STATUS_BUFFER_TOO_SMALL.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_BUFFER_TOO_SMALL

On success, a driver sets **Irp->IoStatus.Information** to the number of bytes written to the buffer at **Parameters.WMI.Buffer**.

Operation

A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver handles WMI IRPs by calling **WmiSystemControl**, that routine calls the driver's *DpWmiQueryReginfo* routine.

If a driver handles an **IRP_MN_REGINFO** request itself, it should do so only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next-lower driver.

Before handling the request, the driver must check **Parameters.WMI.DataPath** to determine whether WMI is

querying registration information (**WMIREGISTER**) or requesting an update (**WMIUPDATE**).

WMI sends this IRP with WMIREGISTER after a driver calls **IoWMIRegistrationControl** with WMIREG_ACTION_REGISTER or WMIREG_ACTION_REREGISTER. In response, a driver should fill in the buffer at **Parameters.WMI.Buffer** with the following:

- A **WMIREGINFO** structure that indicates the driver's registry path, the name of its MOF resource, and the number of blocks to register.
- One**WMIREGGUID** structure for each block to register. If a block is to be registered with static instance names, the driver sets the appropriate WMIREG_FLAG_INSTANCE_*XXX* flag in the **WMIREGGUID** structure for that block.
- Any strings WMI needs to generate static instance names.

WMI sends this IRP with **WMIUPDATE** after a driver calls **IoWmiRegistrationControl** with WMIREG_ACTION_UPDATE_GUIDS. In response, a driver should fill in the buffer at **Parameters.WMI.Buffer** with a WMIREGINFO structure as follows:

- To remove a block, the driver sets WMIREG_FLAG_REMOVE_GUID in its **WMIREGGUID** structure.
- To add or update a block (for example, to change its static instance names), the driver clears WMIREG_FLAG_REMOVE_GUID and provides new or updated registration values for the block.
- To register a new or existing block with static instance names, the driver sets the appropriate WMIREG_FLAG_INSTANCE_*XXX* and supplies any strings WMI needs to generate static instance names.

A driver can use the same **WMIREGINFO** structures to remove, add, or update blocks as it used initially to register all of its blocks, changing only the flags and data for the blocks to be updated. If a **WMIREGGUID** in such a **WMIREGINFO** structure matches exactly the **WMIREGGUID** passed by the driver when it first registered that block, WMI skips the processing involved in updating the block.

WMI does not send an **IRP_MN_REGINFO** request after a driver calls **IoWMIRegistrationControl** with WMIREG_ACTION_DEREGISTER, because WMI requires no further information from the driver. A driver typically deregisters its blocks in response to an **IRP_MN_REMOVE_DEVICE** request.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiQueryReginfo

IoWMIRegistrationControl

WMILIB_CONTEXT

WMIREGGUID

WMIREGINFO

WmiSystemControl

IRP_MN_REGINFO_EX

IRP_MN_REGINFO_EX

6/25/2019 • 5 minutes to read • Edit Online

WMI sends this IRP to query or update a driver's registration information after the driver has called **IoWMIRegistrationControl**. A driver can handle WMI IRPs either by calling **WmiSystemControl** or by handling the IRP itself, as described in Handling WMI Requests.

If a driver calls **WmiSystemControl** to handle an **IRP_MN_REGINFO_EX** request, WMI in turn calls that driver's *DpWmiQueryReginfo* routine.

On Microsoft Windows XP and later operating systems, drivers that support WMI must handle this IRP. Drivers that support Microsoft Windows 98 and Windows 2000 must also handle **IRP_MN_REGINFO**.

Major Code

IRP_MJ_SYSTEM_CONTROL When Sent

On Windows XP and later, WMI sends this IRP to query or update a driver's registration information after the driver has called **IoWMIRegistrationControl**. On Windows 98 and Windows 2000, WMI sends the **IRP_MN_REGINFO** request instead.

WMI sends this IRP at IRQL = PASSIVE_LEVEL in the context of a system thread.

Input Parameters

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath is set to **WMIREGISTER** to query registration information or **WMIUPDATE** to update it.

Parameters.WMI.BufferSize indicates the maximum size of the nonpaged buffer at **Parameters.WMI.Buffer**. The size must be greater than or equal to the total of (**sizeof**(WMIREGINFO) + (*GuidCount* * **sizeof**(WMIREGGUID)), where *GuidCount* is the number of data blocks and event blocks being registered by the driver, plus space for static instance names, if any.

Output Parameters

If the driver handles WMI IRPs by calling **WmiSystemControl**, WMI gets registration information for a driver's data blocks by calling its *DpWmiQueryReginfo* routine.

Otherwise, the driver fills in a **WMIREGINFO** structure at **Parameters.WMI.Buffer** as follows:

- Sets **BufferSize** to the size in bytes of the **WMIREGINFO** structure plus associated registration data.
- If the driver handles WMI requests on behalf of another driver, sets **NextWmiRegInfo** to the offset in bytes from the beginning of this **WMIREGINFO** to the beginning of another **WMIREGINFO** structure that contains registration information from the other driver.
- Sets **RegistryPath** to the registry path that was passed to the driver's **DriverEntry** routine.
- If **Parameters.WMI.Datapath** is set to **WMIREGISTER**, sets **MofResourceName** to the offset from the beginning of this **WMIREGINFO** to a counted Unicode string that contains the name of the driver's MOF

resource in its image file.

- Sets **GuidCount** to the number of data blocks and event blocks to register or update.
- Writes an array of **WMIREGGUID** structures, one for each data block or event block exposed by the driver, at **WmiRegGuid**.

The driver fills in each **WMIREGGUID** structure as follows:

- Sets **Guid** to the GUID that identifies the block.
- Sets **Flags** to provide information about instance names and other characteristics of the block. For example, if a block is being registered with static instance names, the driver sets **Flags** with the appropriate WMIREG_FLAG_INSTANCE_*XXX* flag.

If the block is being registered with static instance names, the driver:

- Sets **InstanceCount** to the number of instances.
- Sets one of the following members to an offset in bytes to static instance name data for the block:
  - If the driver sets **Flags** with WMIREG_FLAG_INSTANCE_LIST, it sets **InstanceNameList** to an offset to a list of static instance name strings. WMI specifies instances in subsequent requests by index into this list.
  - If the driver sets **Flags** with WMIREG_FLAG_INSTANCE_BASENAME, it sets **BaseNameOffset** to an offset to a base name string. WMI uses this string to generate static instance names for the block.
  - If the driver sets **Flags** with WMIREG_FLAG_INSTANCE_PDO, it sets **Pdo** to an offset to a pointer to the PDO passed to the driver's *AddDevice* routine. WMI uses the device instance path of the PDO to generate static instance names for the block. Drivers must call **ObReferenceObject** on the physical device object passed in **Pdo**. The system will automatically call **ObDereferenceObject** to dereference the object; the driver must not do so. (Drivers that use **WmiSystemControl** to handle IRPs do not need to call **ObReferenceObject**. WMI automatically does so before calling the driver's *DpWmiQueryReginfo* routine.)
- Writes the instance name strings, the base name string, or a pointer to the PDO at the offset indicated by **InstanceNameList**, **BaseName**, or **Pdo**, respectively.

If the driver handles WMI registration on behalf of another driver (such as a miniclass or miniport driver), it fills in another **WMIREGINFO** structure with the other driver's registration information and writes it at **NextWmiRegInfo** in the previous structure.

If the buffer at **Parameters.WMI.Buffer** is too small to receive all of the data, a driver writes the needed size in bytes as a ULONG to **Parameters.WMI.Buffer** and fails the IRP and returns STATUS_BUFFER_TOO_SMALL.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_BUFFER_TOO_SMALL

On success, a driver sets **Irp->IoStatus.Information** to the number of bytes written to the buffer at **Parameters.WMI.Buffer**.

Operation

If a driver handles an **IRP_MN_REGINFO_EX** request itself, it should do so only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver must forward the request to the next-lower driver.

Before handling the request, the driver must check **Parameters.WMI.DataPath** to determine whether WMI is querying registration information (**WMIREGISTER**) or requesting an update (**WMIUPDATE**).

WMI sends this IRP with WMIREGISTER after a driver calls **IoWMIRegistrationControl** with WMIREG_ACTION_REGISTER or WMIREG_ACTION_REREGISTER. In response, a driver should fill in the buffer at **Parameters.WMI.Buffer** with the following:

- A **WMIREGINFO** structure that indicates the driver's registry path, the name of its MOF resource, and the number of blocks to register.
- One**WMIREGGUID** structure for each block to register. If a block is to be registered with static instance names, the driver sets the appropriate WMIREG_FLAG_INSTANCE_*XXX* flag in the **WMIREGGUID** structure for that block.
- Any strings WMI needs to generate static instance names.

WMI sends this IRP with **WMIUPDATE** after a driver calls **IoWmiRegistrationControl** with WMIREG_ACTION_UPDATE_GUIDS. In response, a driver should fill in the buffer at **Parameters.WMI.Buffer** with a WMIREGINFO structure as follows:

- To remove a block, the driver sets WMIREG_FLAG_REMOVE_GUID in its **WMIREGGUID** structure.
- To add or update a block (for example, to change its static instance names), the driver clears WMIREG_FLAG_REMOVE_GUID and provides new or updated registration values for the block.
- To register a new or existing block with static instance names, the driver sets the appropriate WMIREG_FLAG_INSTANCE_*XXX* and supplies any strings WMI needs to generate static instance names.

A driver can use the same **WMIREGINFO** structures to remove, add, or update blocks as it used initially to register all of its blocks, changing only the flags and data for the blocks to be updated. If a **WMIREGGUID** in such a **WMIREGINFO** structure matches exactly the **WMIREGGUID** passed by the driver when it first registered that block, WMI skips the processing involved in updating the block.

WMI does not send an **IRP_MN_REGINFO_EX** request after a driver calls **IoWMIRegistrationControl** with WMIREG_ACTION_DEREGISTER, because WMI requires no further information from the driver. A driver typically deregisters its blocks in response to an **IRP_MN_REMOVE_DEVICE** request.

Requirements

Header	Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

See also

DpWmiQueryReginfo

IoWMIRegistrationControl

WMILIB_CONTEXT

WMIREGGUID

WMIREGINFO

WmiSystemControl

IRP_MN_REGINFO

Calling WmiSystemControl to Handle WMI IRPs

6/25/2019 • 2 minutes to read • Edit Online

WMI library routines simplify handling of WMI requests because instead of processing each such request, a driver calls **WmiSystemControl**. In the **WmiSystemControl** call, the driver passes an initialized **WMILIB_CONTEXT** structure that contains entry points to the driver's WMI library callback routines (*DpWmiXxx* routines) and information about the driver's data blocks and event blocks.

Because the WMI library provides no mechanism for passing dynamic instance names or a static instance name list, a driver can use the WMI library to handle requests involving only data blocks with static instance names based on a PDO or a single base name string. For more information about static and dynamic instance names, see Defining WMI Instance Names. Nothing prevents a driver from using the WMI library to handle requests for such blocks and processing requests for other blocks in its *DispatchSystemControl* routine. For more information, see Processing WMI IRPs in a DispatchSystemControl Routine.

To handle WMI IRPs by calling **WmiSystemControl**, a driver must implement certain required *DpWmiXxx* callback routines, and might implement additional optional *DpWmiXxx* callback routines:

- *DpWmiQueryReginfo*—(Required) Provides information about the data and event blocks being registered by the driver. WMI calls a driver's *DpWmiQueryReginfo* routine to process an **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request. For more information, see Using the WMI Library to Register Blocks.
- *DpWmiQueryDataBlock*—(Required) Returns either a single instance or all instances of a data block. WMI calls a driver's *DpWmiQueryDataBlock* routine to process an **IRP_MN_QUERY_SINGLE_INSTANCE** or **IRP_MN_QUERY_ALL_DATA** request.
- *DpWmiSetDataBlock*—(Optional) Changes all data items in a single instance of a data block. WMI calls a driver's *DpWmiSetDataBlock* routine to process an **IRP_MN_CHANGE_SINGLE_INSTANCE** request.
- *DpWmiSetDataItem*—(Optional) Changes a single data item in an instance of a data block. WMI calls a driver's *DpWmiSetDataItem* routine to process an **IRP_MN_CHANGE_SINGLE_ITEM** request.
- *DpWmiFunctionControl*—(Optional) Enables and disables event notification and data collection for blocks registered as expensive to collect. WMI calls a driver's *DpWmiFunctionControl* routine to process an **IRP_MN_ENABLE_COLLECTION**, **IRP_MN_DISABLE_COLLECTION**, **IRP_MN_ENABLE_EVENTS**, or **IRP_MN_DISABLE_EVENTS** request.
- *DpWmiExecuteMethod*—(Optional) Executes a method associated with a data block. WMI calls a driver's *DpWmiExecuteMethod* routine to process an **IRP_MN_EXECUTE_METHOD** request.

A driver's *DpWmiXxx* routines can have any names chosen by the driver writer.

Before calling **WmiSystemControl**, the driver must initialize a **WMILIB_CONTEXT** structure with entry points to its *DpWmiXxx* routines and information about its data blocks and event blocks.

When the driver receives a WMI request:

1. The driver calls **WmiSystemControl** with a pointer to its initialized **WMILIB_CONTEXT** structure, a pointer to its device object, and a pointer to the IRP.
2. WMI validates the IRP parameters and calls the driver's *DpWmiXxx* routine that processes the request. If the driver set no entry point in its **WMILIB_CONTEXT** for an optional *DpWmiXxx* routine, WMI completes the IRP with default values and status.

3. In its *DpWmiXxx* routine, the driver processes the request and writes any output to the caller-supplied buffer. For example, a driver's *DpWmiQueryDataBlock* routine would write the requested instance(s) of the specified block to the buffer.

4. In all *DpWmiXxx* routines except *DpWmiQueryReginfo*, the driver calls **WmiCompleteRequest** to complete the request, or returns STATUS_PENDING to postpone completion, as for any IRP.

5. WMI performs any necessary postprocessing, packages any output in an appropriate **WNODE_*XXX*** structure, and passes the output and status to the data consumer.

Processing WMI IRPs in a DispatchSystemControl Routine

6/25/2019 • 2 minutes to read • Edit Online

A driver that handles WMI IRPs in its *DispatchSystemControl* routine must handle such an IRP only if the device object pointer at **Parameters.WMI.ProviderId** matches the pointer passed by the driver in its call to **IoWMIRegistrationControl**. Otherwise, the driver must forward the IRP to the next lower driver.

If the driver handles the request, it must:

Check the GUID at **Parameters.WMI.DataPath** to determine whether it represents a data block supported by the driver and, if not, fail the IRP with STATUS_WMI_GUID_NOT_FOUND.

A driver should check the input **WNODE_*XXX*** structure at **Parameters.WMI.Buffer** for the instance name when handling any of the following requests:

IRP_MN_QUERY_SINGLE_INSTANCE IRP_MN_CHANGE_SINGLE_INSTANCE IRP_MN_CHANGE_SINGLE_ITEM IRP_MN_EXECUTE_METHOD The driver should check for the instance name as follows:

- If WNODE_FLAG_STATIC_INSTANCE_NAMES is set in **WnodeHeader.Flags**, use **InstanceIndex** as an index into the driver's list of static instance names for that block.
- If WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags**, use **OffsetInstanceName** as an offset to the instance name string in the input **WNODE_*XXX*** structure. **OffsetInstanceName** is the offset in bytes from the beginning of the structure to a USHORT that indicates the length of the instance name string in bytes (not characters), including the NUL terminator if present, followed by the string itself in Unicode.

If the driver cannot locate the instance specified by **InstanceIndex** or **OffsetInstanceName**, it must fail the IRP with STATUS_WMI_INSTANCE_NOT_FOUND.

For an **IRP_MN_EXECUTE_METHOD** request, check **MethodID** in the input **WNODE_METHOD_ITEM** and, if the method is not valid for that data block, fail the IRP with STATUS_WMI_ITEMID_NOT_FOUND.

If the request generates output, a driver should check the size of the buffer at **Parameters.WMI.BufferSize** when handling any of the following requests:

IRP_MN_QUERY_ALL_DATA IRP_MN_QUERY_SINGLE_INSTANCE IRP_MN_EXECUTE_METHOD If the buffer is too small to receive the output, but at least **sizeof**(**WNODE_TOO_SMALL**), the driver should succeed the IRP and write a **WNODE_TOO_SMALL** structure to the buffer at **Parameters.WMI.Buffer**. If the buffer is smaller than **sizeof**(**WNODE_TOO_SMALL**), the driver fails the IRP with an NTSTATUS code of STATUS_BUFFER_TOO_SMALL.

If the request generates output and the buffer size is adequate, write the following output to the buffer at **Parameters.WMI.Buffer**:

- For an **IRP_MN_QUERY_ALL_DATA** request, the driver writes a **WNODE_ALL_DATA** structure that contains data for all instances of the specified data block.
- For an **IRP_MN_QUERY_SINGLE_INSTANCE** request, the driver writes a **WNODE_SINGLE_INSTANCE** structure that contains data for the specified instance of a data block.
- For an **IRP_MN_EXECUTE_METHOD** if the method generates output, the driver writes the method output in driver-determined format following the input **WNODE_METHOD_ITEM** in the buffer (overwriting input data,

if any).

Set **Irp->IoStatus.Information** to the number of bytes written to the buffer at **Parameters.WMI.Buffer** and **Irp->IoStatus.Status** to STATUS_SUCCESS.

Call **IoCompleteRequest** to complete the IRP.

For more information, see WMI WNODE_*XXX* Structures.

WMI WNODE_XXX Structures

6/25/2019 • 5 minutes to read • Edit Online

WMI uses a set of standard data structures called **WNODE_*XXX*** to pass data between user-mode data consumers and kernel-mode data providers such as drivers. If a driver handles WMI requests by calling **WmiSystemControl**, the driver is not required to read or write **WNODE_*XXX*** structures. Otherwise, the driver must interpret the input **WNODE_*XXX*** at **Parameters.WMI.Buffer** and/or write an output **WNODE_*XXX*** to that location.

The following table lists WMI IRPs and their corresponding **WNODE_*XXX*** structures.

WMI IRP	RELATED WNODE_XXX STRUCTURE
IRP_MN_CHANGE_SINGLE_INSTANCE	WNODE_SINGLE_INSTANCE
IRP_MN_CHANGE_SINGLE_ITEM	WNODE_SINGLE_ITEM
IRP_MN_EXECUTE_METHOD	WNODE_METHOD_ITEM
IRP_MN_QUERY_ALL_DATA	WNODE_ALL_DATA
IRP_MN_QUERY_SINGLE_INSTANCE	WNODE_SINGLE_INSTANCE

Two additional **WNODE_*XXX*** structures, **WNODE_EVENT_ITEM** and **WNODE_EVENT_REFERENCE**, are used to send notifications of enabled events. A driver that registers event blocks will, if an event is enabled and the event occurs, send notification of the event to WMI by calling **IoWMIWriteEvent** and passing a **WNODE_EVENT_*XXX*** structure. For information about sending WMI events, see Sending WMI Events.

Each **WNODE_*XXX*** structure consists of the following:

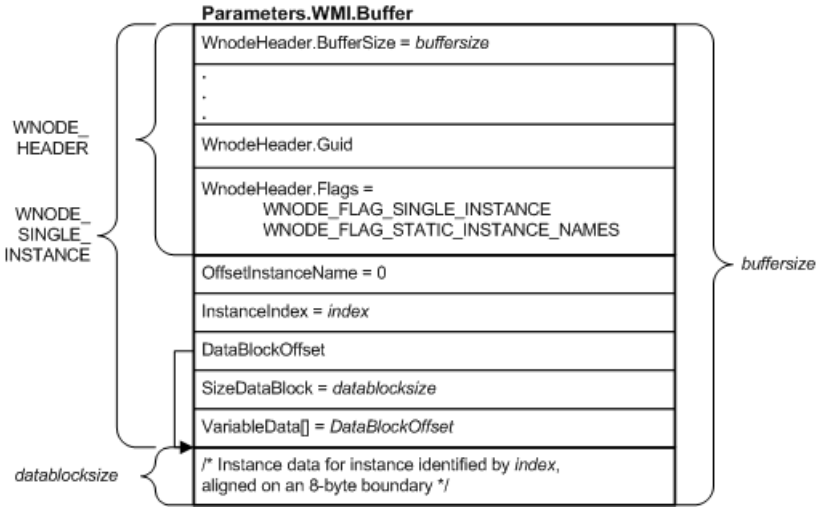
- An embedded **WNODE_HEADER** structure that contains information common to all **WNODE_*XXX*** including the size of the buffer, the GUID that represents the data block, and flags that indicate the type of **WNODE_*XXX*** structure, whether it uses static or dynamic instance names, and other characteristics of the block.
- The fixed members of the particular **WNODE_*XXX*** structure, such as offsets to instance names and data.

A **WNODE_*XXX*** structure in an IRP buffer (**Parameters.WMI.Buffer**) is typically followed by variable data related to the request, such as dynamic instance names, static instance name strings, input for or output from a method, or data for one or more instances of a data block. The size of the buffer must therefore exceed **sizeof(WNODE_*XXX*)** by the amount of variable data involved.

Note that WMI does not perform type-checking on variable data supplied by a driver. The driver must align output data on an appropriate boundary in the output buffer so that a data consumer can parse the data correctly. In particular, each instance must start on an 8-byte boundary and each of its items must be aligned on a natural boundary according to the data block schema previously registered by the driver. Dynamic instance names can be aligned on a 2-byte boundary.

The following figure shows a block diagram of an IRP buffer containing a **WNODE_SINGLE_INSTANCE**

structure that a driver might return in response to an **IRP_MN_QUERY_SINGLE_INSTANCE** request.



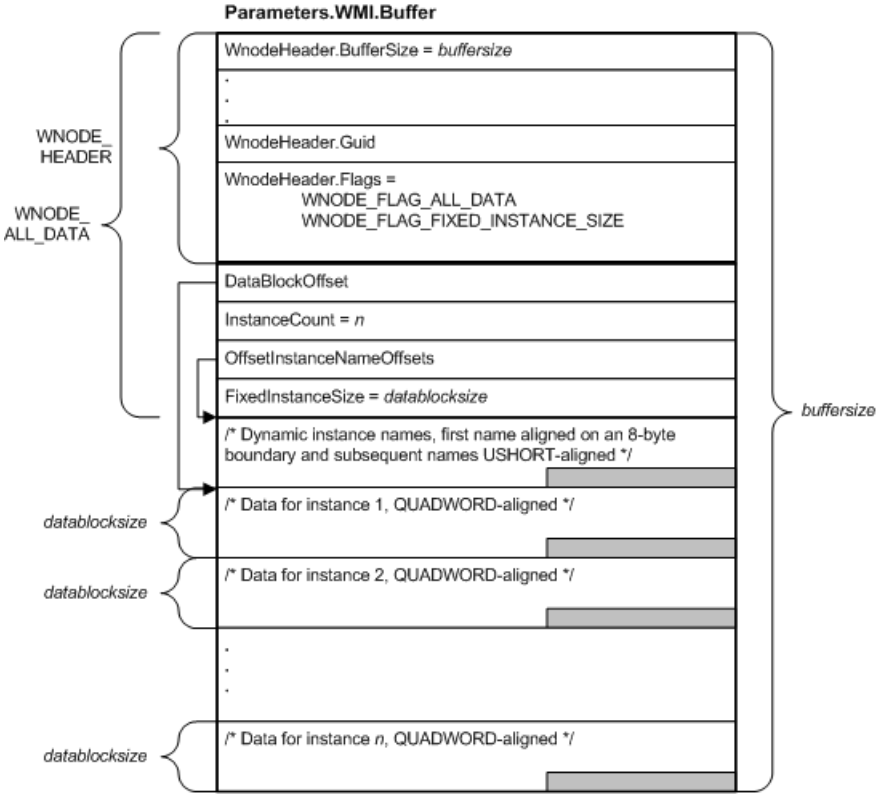
Starting at the top of the previous figure:

- The **WNODE_HEADER** structure at the beginning of the **WNODE_SINGLE_INSTANCE** is contained in a **WnodeHeader** member. WMI fills in all members of the **WNODE_HEADER** before sending the request. In the **WNODE_HEADER**:
  - **WnodeHeader.Buffersize** indicates the size of the **WNODE_SINGLE_INSTANCE**, including data that follows the fixed members of the structure. (The value of **WnodeHeader.Buffersize** is typically less than **Parameters.WMI.Buffersize**, which indicates the size of the buffer allocated by WMI to receive output from the driver.)
  - **WnodeHeader.Guid** contains the GUID that identifies the data block.
  - In this example, **WnodeHeader.Flags** indicates that this structure is a **WNODE_SINGLE_INSTANCE** and that the data block uses static instance names.
- Because the data block uses static instance names, WMI sets **InstanceIndex** to the index of the instance in the list of static instance names passed by the driver when it registered the block. **OffsetInstanceNames** is not used.
- WMI sets **DataBlockOffset** to indicate the offset from the beginning of the buffer to the first byte of instance data. (The driver must not change this value) Again because the data block uses static instance names, this offset indicates the same location as **VariableData**. If the data block used dynamic instance names, the instance names would start at **VariableData** and **DataBlockOffset** would specify a greater offset into the buffer.
- The driver sets **SizeDataBlock** to the number of bytes of instance data being returned.
- At **VariableData** (after instance name data, if present), the driver writes instance data for the requested instance in the output buffer.

A driver reads and writes **WNODE_METHOD_ITEM** and **WNODE_SINGLE_ITEM** structures in much the same way as **WNODE_SINGLE_INSTANCE**. These structures resemble each other in that each has the fixed members **OffsetInstanceName**, **InstanceIndex**, **DataBlockOffset**, **SizeDataBlock** (or, in the case of **WNODE_SINGLE_ITEM**, **SizeDataItem**) and **VariableData**. **WNODE_METHOD_ITEM** includes a **MethodId** and **WNODE_SINGLE_ITEM** includes an **ItemId** which **WNODE_SINGLE_INSTANCE** lacks.

WNODE_ALL_DATA differs from the preceding structures in that it is used to pass multiple instances of a data block, possibly including dynamic instance names and possibly of different sizes.

The following figure shows a block diagram of an IRP buffer containing a **WNODE_ALL_DATA** that a driver might return in response to an **IRP_MN_QUERY_ALL_DATA** request.



Starting at the top of the previous figure:

- As described in the previous figure, the **WNODE_HEADER** structure at the beginning of the **WNODE_ALL_DATA** is contained in a **WnodeHeader** member. **WnodeHeader.Buffersize** and **WnodeHeader.Guid** indicate the size of the **WNODE_ALL_DATA** and the GUID of the data block, respectively.

  In this example, WMI sets **WnodeHeader.Flags** to indicate that this structure is a **WNODE_ALL_DATA** and that the data block was registered with dynamic instance names (that is, WMI clears WNODE_FLAG_STATIC_INSTANCE_NAMES and WNODE_FLAG_PDO_INSTANCE_NAMES). On output, the driver sets WNODE_FLAG_FIXED_INSTANCE_SIZE to indicate that all of the instances are the same size.

- WMI sets **DataBlockOffset** to indicate the offset from the beginning of the buffer to the first byte of instance data. (The driver must not change this value). In this example, instance data follows the instance names at **OffsetInstanceNameOffsets**.

- The driver sets **InstanceCount** to indicate the number of instances being returned.

- **WNODE_*XXX*** for data blocks that use dynamic instance names always contain the instance name strings. Because this example uses dynamic instance names, **OffsetInstanceNameOffsets** indicates the offset from the beginning of the buffer to an array of offsets to dynamic instance names in the buffer.

- **FixedInstanceSize** indicates the number of bytes of data in each instance being returned by the driver. If instances of this data block were to vary in size, the driver would clear WNODE_FLAG_FIXED_INSTANCE_SIZE in **WnodeHeader.Flags** and set **OffsetInstanceDataAndLength** to an array of **OFFSETINSTANCEDATAANDLENGTH** structures, each specifying an offset to the data for one instance and the number of bytes in that instance instead of setting **FixedInstanceSize**.

For more information about **WNODE_*XXX*** structures, see System Structures.

Sending WMI Events

6/25/2019 • 2 minutes to read • Edit Online

A driver can use WMI events to notify user-mode applications of events without requiring the applications to poll or send IRPs. A driver should use WMI events to notify WMI clients of exceptional conditions, not as an alternative to error logging. A driver should support any standard event blocks defined for its device type in Wmicore.mof, and might define and register additional custom event blocks to support device-specific notifications.

An event block is simply a data block that derives from the abstract base class **WMIEvent**. An event block can contain any of the same data as a data block, or it can be empty—that is, an event block need not contain any driver-defined data items. If an event block does contain data, the total size of the **WNODE_*XXX*** plus the data should not exceed the registry-defined limit of 1 kilobyte. In general, smaller events result in better system performance and more timely notification. For information about defining blocks, see MOF Syntax for WMI Data and Event Blocks and Designing WMI Data and Event Blocks.

A driver indicates support for an event by registering the corresponding event block with WMIREG_FLAG_EVENT_ONLY_GUID set in the block's **WMIREGGUID** structure. For information about registering blocks, see Registering as a WMI Data Provider.

When a WMI client user requests notification of an event, WMI sends an **IRP_MN_ENABLE_EVENTS** request to the driver, which alerts the driver to begin monitoring the event's driver-determined trigger condition. Then, when the trigger condition occurs, the driver sends the event to WMI, which delivers it to all data consumers that have registered for the event.

A driver sends an event to WMI in one of the following ways:

- Call the kernel-mode WMI library routine **WmiFireEvent**. A driver can call **WmiFireEvent** to send only events that do not use dynamic instance names, and that base static instance names on a single base name string or the device instance ID of a PDO. Furthermore, the event must be a single instance—that is, a driver cannot call **WmiFireEvent** to send an event that consists of a single item or multiple instances. For more information, see Sending an Event with WmiFireEvent.
- Call the kernel-mode routine **IoWMIWriteEvent** with a pointer to a driver-allocated and initialized **WNODE_*XXX*** structure that contains the event's data. For more information, see Sending an Event with IoWMIWriteEvent.

Sending an Event with WmiFireEvent

6/25/2019 • 2 minutes to read • Edit Online

A driver can call **WmiFireEvent** to send events that do not use dynamic instance names, and that base static instance names on a single base name string or the device instance ID of a PDO.

The event must be a single instance of a block—that is, a driver cannot call **WmiFireEvent** to send an event that consists of a single item or multiple instances. To send such events, a driver must call **IoWMIWriteEvent**, as described in Sending an Event with IoWMIWriteEvent.

A driver should not send events until WMI has enabled the event. After the event has been enabled, when the event's trigger condition occurs, the driver:

1. Allocates a buffer from the nonpaged pool and writes the event data to the buffer. If the event has no data, the driver can skip this step.
2. Calls **WmiFireEvent** with the following parameters:

   - A pointer to the driver's device object
   - A pointer to the GUID that represents the event block
   - If the event block has multiple instances, the index of the instance
   - If data is to be sent with the event, the number of bytes of data, or 0 if none
   - If data is to be sent with the event, a pointer to the driver-allocated buffer that contains the data, or **NULL** if none

   The driver must allocate all parameters passed to **WmiFireEvent**, including the event data buffer, from nonpaged pool. WMI releases the driver-allocated memory without further intervention by the driver.

After **WmiFireEvent** returns, the driver resumes monitoring the event's trigger condition and sends the event each time its trigger condition occurs until WMI disables that event.

Sending an Event with IoWMIWriteEvent

6/25/2019 • 2 minutes to read • Edit Online

A driver can call **IoWMIWriteEvent** to send any event. The event can consist of a single item, a single instance, or all instances of a data block, and it can use dynamic instance names.

Unlike **WNODE_*XXX*** structures passed with query or change requests, which are allocated and partially initialized by WMI, the driver must allocate and initialize all members of the **WNODE_*XXX*** structure that contains an event.

A driver must send an event only after WMI has sent an **IRP_MN_ENABLE_EVENTS** request to enable the event. Then, when the event's trigger condition occurs, the driver:

1. Allocates a buffer from nonpaged pool to contain the **WNODE_*XXX*** structure needed for the event, including space for variable data, if any.

   Depending on the event, the driver might allocate a **WNODE_SINGLE_ITEM**, a **WNODE_SINGLE_INSTANCE**, or a **WNODE_ALL_DATA** for the event. The size of the **WNODE_*XXX*** plus variable data must not exceed the registry-defined limit of 1K.

2. Initializes all members of the **WNODE_*XXX*** structure, including **WnodeHeader.Flags**:

   - The driver sets the **WNODE_FLAG_EVENT_ITEM** flag to indicate that the structure is an event.
   - The driver sets one of the following flags to indicate the type of **WNODE_*XXX*** structure:

     WNODE_FLAG_ALL_DATA

     WNODE_FLAG_SINGLE_INSTANCE

     WNODE_FLAG_SINGLE_ITEM

   - The driver sets or clears the following flags to indicate whether the block uses static or dynamic instance names:

     WNODE_FLAG_STATIC_INSTANCE_NAMES

     WNODE_FLAG_PDO_INSTANCE_NAMES

   - The driver might set additional flags depending on the event.

3. Casts a pointer to the **WNODE_*XXX*** to a PWNODE_EVENT_ITEM.

4. Calls **IoWMIWriteEvent** with the pointer.

   If **IoWMIWriteEvent** completes successfully, WMI releases the driver-allocated memory for the event.

After **IoWMIWriteEvent** returns, the driver resumes monitoring the event's trigger condition and sending the event each time its trigger condition occurs, until WMI sends an **IRP_MN_DISABLE_EVENTS** request to disable that event.

If the size of an event exceeds the registry-defined maximum of 1K (not recommended) the driver should call **IoWmiWriteEvent** with an initialized **WNODE_EVENT_REFERENCE** that specifies the event's GUID, its size, and its instance index (for static instance names) or name (for dynamic instance names). WMI will use the information in the **WNODE_EVENT_REFERENCE** to query for the event.

A driver can send an events that does not use dynamic instance names and that consists of a single instance by

calling the WMI library routine **WmiFireEvent**. The driver does not need to allocate and initialize a **WNODE_*XXX*** structure for a **WmiFireEvent** call. WMI packages the driver's event data in a **WNODE_SINGLE_INSTANCE** and delivers it to data consumers. For more information about sending events with **WmiFireEvent**, see Sending an Event with WmiFireEvent.

Using Custom WMI Events

12/5/2018 • 2 minutes to read • Edit Online

Some classes of drivers are required to support certain WMI event classes. Drivers can also design their own custom WMI event classes. Custom WMI events offer a way for a driver to pass data back to a user-mode component. A user-mode component receives WMI events through WMI COM interfaces.

An application can receive event notifications as follows:

- Use the **CoCreateInstance** routine to get a pointer to an **IWbemLocator** object.
- Use the **IWbemLocator** pointer to connect to the WMI server process. The **IWBemLocator::ConnectServer** method call provides you with a pointer to an **IWbemServices** object.
- Use the **IWbemServices** object to query for the event types you are interested in. The **IWbemServices::ExecNotificationQuery** method allows you to specify an event query in the WMI Query Language (WQL).
- An application can also register to receive WMI events asynchronously, by implementing the **IWbemObjectSink** interface. The application uses the **IWbemServices::ExecNotificationQueryAsync** method to register for asynchronous notification of events. When matching events occur, the system uses the **IWbemObjectSink::Indicate** method to notify the application of the events that have occurred.

You can also implement a user-mode WMI *event consumer provider*. This is a user-mode component that WMI can automatically load when events of a specified type occur.

- Include an instance of the **__EventConsumerProviderRegistration** WMI class in the MOF data for your user-mode component.
- Implement the **IWbemUnboundObjectSink** interface for each WMI event class you want to receive notifications of.
- Implement the **IWbemEventConsumerProvider** interface to specify the event classes the component receives notifications of, and the associated **IWbemUnboundObjectSink** implementations.
- Implement the **IWbemProviderInit** interface that initializes your component as an event consumer.

More information about receiving WMI events and the **IWbemXxx** COM interfaces can be found in the Microsoft Windows SDK documentation.

WMI events are not the only way to notify user-mode applications when particular situations occur. A driver could implement an IOCTL that an application could use to poll for notification. The driver and application could share a notification event object (see Event Objects) to signal that a particular situation has occurred.

WMI events have some advantages over these other methods:

- If user-mode applications poll for events faster than the driver can respond, then the driver may have many IOCTLs pending.
- You can ameliorate the previous problem by using a notification event object to notify a user-mode application, but notification events can only signal that an event has occurred. The application must still use an IOCTL to get any additional data. The next two issues still apply.
- If multiple applications poll the driver for events, the driver would need to maintain state to determine which applications had received which events.

- Some drivers, such as SCSI miniport and NDIS miniport drivers, cannot receive IOCTLs.

WMI events do have the disadvantage that the user-mode code you must provide is considerably more complicated than that for the other methods.

WMI Property Sheets

6/25/2019 • 2 minutes to read • Edit Online

A user-friendly driver allows users to control its settings through its **Device Manager** property sheet. See Using Device Manager for a description of Device Manager.

Drivers can automatically expose any WMI classes they implement on their property sheet by using the WMI generic property page provider.

Drivers can enable certain controls on the **Power Management** tab of the **Device Manager** property sheet by supporting certain particular WMI class GUIDs. See WMI and the Power Management Tab for details.

WMI Generic Property Page Provider

6/25/2019 • 2 minutes to read • Edit Online

On Windows XP and later operating systems, drivers can expose their WMI classes through the WMI generic property page provider. The provider uses each class declaration to create a simple property page for the class properties.

How Property Qualifiers Determine the Property Page

The WMI generic property page provider uses a control appropriate for the data type of each property in the class. The following property qualifiers modify the type of control used:

- **Write**

  A property with the **write** qualifier can be changed through the property page. Otherwise the property is read-only.

- **Values** and **ValuesMap**

  The generic property page provider uses a list box to represent the possible values.

- **Range**

  The generic property page provider validates that the data entered conforms to the specified range.

- **DisplayName**

  The generic property page provider uses the value of this property qualifier as the label for the property.

- **DisplayInHex**

  If present, the property value is displayed in hexadecimal.

Driver writers should localize property qualifiers that are strings. See Localizing MOF Files for details.

Enabling the Generic Property Page Provider

Each device that exposes classes to be used by Wmiprop.dll must enable Wmiprop.dll as a co-installer. To do this, make the following addition to the co-installer *add-registry-section*: add a value entry for the class GUID under the **HKLM\System\CurrentControlSet\Control\CoDeviceInstallers** registry key. The value for the value entry is "WmiProp.dll, WmiPropCoInstaller".

For example:

```
; This section is defined in the Co-installer section, as follows.
; [Co-installer]
; AddReg = CoInstaller_AddReg

[CoInstaller_AddReg]
HKLM, System\CurrentControlSet\Control\CoDeviceInstallers, ClassGUID,
    0x00010000, "WmiProp.dll, WmiPropCoInstaller"
```

ClassGUID is the GUID for the WMI class. See Registering a Class Co-installer for details.

You must also specify the particular WMI classes to be exposed through the generic property provider. To do this, set the **WmiConfigClasses** value-entry to be a comma-separated list of the WMI classes in the *add-registry-section* of the device class or device hardware instance.

```
; the device class AddReg section.
[device_class_AddReg]
HKR,,"WmiConfigClasses",0x00000000,"class1,class2"

; the device hardware instance AddReg section.
[device_hw_inst_AddReg]
HKR,,"WmiConfigClasses",0x00000000,"class3"
```

See **INF AddReg Directive** for a description of an *add-registry-section* in INF files.

Wmiprop.dll assumes only one instance of each class. Each class is represented by a tab on the property sheet. Use the **DisplayName** property qualifier to set the title text of the tab. A property page for a class only appears if there is currently an instance of the class. Therefore, if the device is removed or not started, the pages do not appear.

WMI and the Power Management Tab

6/25/2019 • 2 minutes to read • Edit Online

Drivers that support power management can automatically enable the **Power Management** tab for the device property sheet in Device Manager. If a driver handles the GUID_POWER_DEVICE_ENABLE or GUID_POWER_DEVICE_WAKE_ENABLE WMI class GUIDs, Device Manager displays a **Power Management** tab on the device property sheet. Certain controls on the property page are enabled depending on which WMI class GUIDs the driver supports.

The GUID_POWER_DEVICE_*XXX* class GUIDs enable controls on the property page as follows:

- GUID_POWER_DEVICE_ENABLE

  Enables a check box to activate or deactivate power management for the device. The data block for the WMI class consists of a single BOOLEAN value that indicates whether power management is enabled. The meaning of the value is device-dependent.

- GUID_POWER_DEVICE_WAKE_ENABLE

  Enables a check box to activate or deactivate sending wait/wake IRPs. When selected, the driver should send an **IRP_MN_WAIT_WAKE** request to its physical device object. This enables the device to wake the system in response to an external event. For example, when enabled for the keyboard class driver, the keyboard device will wake the system when a key is pressed. When the check box is not selected, the driver should cancel the **IRP_MN_WAIT_WAKE** request. The data block for the WMI class consists of a single BOOLEAN value that indicates the current state of the check box.

WMI query requests are sent for the GUID_POWER_DEVICE_*XXX* WMI class GUIDs whenever the property sheet for the driver is opened in Device Manager. The WMI change requests are sent whenever one of the check box values on the **Power Management** tab changes. Users will expect the value they set to persist between driver loads and unloads, so drivers should store the current value of either property in the registry.

The mouse or keyboard class sample drivers both handle the GUID_POWER_DEVICE_WAKE_ENABLE WMI class GUID. See \src\input\kbdclass and \src\input\mouclass in the Windows Driver Kit (WDK).

Using Wmimofck.exe

12/5/2018 • 2 minutes to read • Edit Online

Included with the Windows Driver Kit (WDK) is the Wmimofck.exe utility. This application takes as input a binary MOF file (a .bmf file), which was generated by the MOF compiler (mofcomp.exe). Wmimofck.exe will check that the classes, properties, methods and events specified in the .bmf file are valid for WMI use. Wmimofck.exe is also capable of generating the following files:

- C language header file (.h file) that can then be used to keep header file in sync with MOF definitions.
- C language source file which contains stubs for WMI driver code.
- Hex version of .bmf data which can be included in the driver source for supplying dynamic MOF data at runtime.
- Test application templates in VBScript or HTML.

To run the **wmimofck** utility, use the following syntax:

wmimofck [**-h***filename* [**-m**] [**-u**]] [**-c***filename*] [**-x***filename*] [**-t***filename*] [**-w***directory*] [*-yfilename*] [*-zfilename*]

If the **-h** parameter is specified, a C language header file is created that defines the GUIDs, data structures, and method indices specified in the MOF file. If the caller specifies the **-m** flag as well, then the header file will include structure definitions for the input and output of each WMI method. By default, *wmimofck* does not generate member definitions for WMI classes that contain variable length properties. If the caller specifies **-u**, then *wmimofck* will generate member definitions for every property that has a fixed size, including string properties that specify a **MaxLen** qualifier. If the **-t** parameter is specified, a VBScript program is created that will query all data blocks and properties specified in the MOF file.

If the **-x** parameter is specified a text file is created that contains the text representation of the binary MOF data. This can be included in the source of the driver if the driver supports reporting the binary MOF via a WMI query rather than a resource on the driver image file.

If the **-c** parameter is specified, a C language source file is generated that contains a template for implementing WMI code in a device driver.

If the **-w** parameter is specified, a set of HTML files are generated that create a rudimentary UI that can be used to access the WMI data blocks.

The **-y** and **-z** flags can only be used together. The **-y** specifies a file containing language-independent WMI class declarations, and **-z** specifies the class amendments for a particular language. The command *wmimofck localizedfile* **-y***mof* **-z***mfl* merges the *mof* and *mfl* files to form the complete localized version of MOF file. See Building and Deploying the Localized MOF File for details.

WMI Event Tracing

6/25/2019 • 3 minutes to read • Edit Online

This section describes the WMI extensions to WDM (supported by Windows 2000 and later) that kernel-mode drivers, as information providers, can use to provide information to information consumers. Drivers typically provide information that a consumer uses to determine the driver's configuration and resource usage. In addition to the WMI extensions to WDM, a user-mode API supports providers or consumers of WMI event information—see the Windows SDK for more information.

The event tracing logger supports up to 32 instances. One of the instances is reserved for tracing the kernel. The logger supports tracing a high event rate.

Trace events are defined in the same manner as other WMI events. WMI events are described in the MOF file. For more information about WMI event descriptions, see MOF Syntax for WMI Data and Event Blocks.

The process by which kernel-mode drivers log information is integrated into the existing WMI infrastructure. To log trace events, a driver does the following:

1. Register as a WMI provider by calling **IoWMIRegistrationControl**.
2. Mark events as traceable by setting WMIREG_FLAG_TRACED_GUID in the **Flags** member of the **WMIREGGUID** structure that is passed when the driver registers events with WMI.
3. Specify one event as the control event for overall enabling/disabling of a set of trace events by setting WMIREG_FLAG_TRACE_CONTROL_GUID in the **Flags** member of the **WMIREGGUID** structure that is passed when the driver registers events with WMI.
4. Upon receiving a request from WMI to enable events where the GUID matches the trace control GUID, the driver should store the handle to the logger. The value will be needed when writing an event. For information about how to use this handle, see step 6. The logger handle value is contained in the **HistoricalContext** member of the **WNODE_HEADER** portion of the WMI buffer that is part of the parameters in the enable events request.
5. Decide whether the trace event will be sent to WMI event consumers or is targeted for the WMI event logger only. This will determine where the memory for the **EVENT_TRACE_HEADER** structure should come from. This memory will eventually be passed to **IoWMIWriteEvent**.

   If the event is a log event only, the memory will not be deleted by WMI. In this case, the driver should pass in a buffer on the stack or should be reusing an allocated buffer for this purpose. For performance reasons, the driver should minimize any unnecessary calls to allocate or free memory. Failure to comply with this recommendation will compromise the integrity of the timing information contained in the log file.

   If the event is to be sent to both the logger and to WMI event consumers, then the memory must be allocated from a nonpaged pool. In this case the event will be sent to the logger and then forwarded to WMI to be sent to WMI event consumers who have requested notification of the event. The memory for the event will then be freed by WMI according to the behavior of **IoWMIWriteEvent**.
6. After the memory for the **EVENT_TRACE_HEADER** and any driver event data, if any, has been secured, the following information should be set:

   Set the **Size** member to the sizeof(**EVENT_TRACE_HEADER**) plus the size of any additional driver event data that will be appended on to the end of **EVENT_TRACE_HEADER**.

   Set the **Flags** member to WNODE_FLAG_TRACED_GUID to have the event sent to the logger. If the event

is to be sent to WMI event consumers as well, set the WNODE_FLAG_LOG_WNODE. Note, it is not necessary to set WNODE_FLAG_TRACED_GUID if setting WNODE_FLAG_LOG_WNODE. If both are set, WNODE_FLAG_TRACED_GUID will take precedence and the event will not be sent to WMI event consumers.

Set the **Guid** or the **GuidPtr** member. If using **GuidPtr**, set WNODE_FLAG_USE_GUID_PTR in the **Flags** member.

Optionally, specify a value for **TimeStamp**. If the driver does not specify a **TimeStamp** value the logger will fill this in. If the driver does not want the logger to set the time stamp then it should set WNODE_FLAG_USE_TIMESTAMP in the **Flags** member.

Set any of the following **EVENT_TRACE_HEADER** members that have meaning to the driver: **Class.Type**, **Class.Level**, and **Class.Version**.

Finally cast the **EVENT_TRACE_HEADER** to a **WNODE_HEADER** and set the **HistoricalContext** value of the **Wnode** to the logger handle that was saved in step 4 above.

7. Call **IoWMIWriteEvent** with the pointer to the **EVENT_TRACE_HEADER** structure.

The driver should continue logging trace events associated with the control GUID until the driver receives notification to disable event logging via an **IRP_MN_DISABLE_EVENTS** request.

General Techniques for Testing WMI Driver Support

6/25/2019 • 2 minutes to read • Edit Online

WMI Client Tools

There are several tools you can use to test WMI support in your driver.

Wbemtest
The operating system includes the Wbemtest tool, which provides a GUI you can use to query for WMI classes and class instances, change property values, execute methods, and receive event notifications. Connect to the "root\wmi" namespace to test your driver's support.

Wmic
Microsoft Windows XP and later operating systems include the Wmic tool, which provides a command shell you can use to issue WMI-related commands to test your driver.

Wmimofck
The **wmimofck** command can be used to check the syntax of your binary MOF files. You can also use the **wmimofck -t** command to generate a VBScript file. You can use this script to test your driver's handling of WMI class instance queries. The **wmimofck -w** command generates webpages that can test querying and setting classes, executing methods, and receiving events. Note that the webpages do not support executing methods that use complex parameters or return values (such as an array of embedded classes). In such cases you can use Wbemtest instead. See Using wmimofck.exe for more information about Wmimofck.

You can also test your driver's WMI support by writing a custom WMI client application, using the WMI user-mode API.

For more information about this user-mode API, which allows applications to provide or consume WMI information, refer to the Windows Management Instrumentation information in the Microsoft Windows SDK documentation.

A WMI client application performs the following tasks to test a driver:

- Connects to WMI.

  To connect to WMI, the application can call the Component Object Model (COM) function, **CoCreateInstance**, to retrieve a pointer to the **IWbemLocator** interface. The application then calls the **IWbemLocator::ConnectServer** method to connect to WMI. From this call, the application receives a pointer to the **IWbemServices** interface.

- Accesses information in the driver.

  To access information and to register for events, the application uses the methods of the **IWbemServices** interface.

WMI IRPs and the System Event Log

WMI errors that occur strictly in kernel-mode are logged to the system event log. You can use the Event Viewer to examine the system event log. (See Logging Errors for more information.)

The two main sources of such errors are malformed replies to WMI requests and incorrect parameters to event notifications. For example, if the driver returns a malformed **WMIREGINFO** data structure in response to an **IRP_MN_REGINFO** or **IRP_MN_REGINFO_EX** request, the system will log that to the system event log. The system would also log an invalid call to **IoWMIWriteEvent** and **WmiFireEvent** to issue a WMI event notification.

WMI WDM Provider Log

WMI errors that occur while being handled by the WMI WDM provider (Wmiprov.dll) are logged to the log file for the WMI WDM Provider, Wmiprov.log. This is a text file can be found in %windir%\system32\wbem\logs\wmiprov.log. Errors, such as a bad or missing MOF resource for the driver, are logged here. In the case of a bad MOF resource, the file %windir%\system32\mofcomp.log might have additional information related to the error.

In versions of Windows earlier than Windows Vista, you can change the logging settings for all WMI providers by using the Wmimgmt.msc application. (In Windows 98/Me, use Wbemcntl instead.) You can disable or reenable logging, change the directory where WMI log files are kept, as well as set the maximum size for such files. For more information, see WMI Log Files.

Troubleshooting Specific WMI Problems

6/25/2019 • 3 minutes to read • Edit Online

Driver's WMI Classes Do Not Appear in the \root\wmi Namespace

1. Use wmimofckdriver.bmf to check if the binary MOF file format is correct. Additional error messages may be found in mofcomp.log.
2. Check the system event log to see if the driver is returning a malformed **WMIREGINFO** data structure in response to the registration request.
3. Check that the driver is returning the correct values for **RegistryPath** and **MofResourceName** within the **WMIREGINFO** structure.
4. If the driver provides its MOF data in a separate file, check that the MofImagePath registry value for the driver is set correctly.
5. Check the WMI WDM provider log for errors.
6. Use Mofcomp to recompile and reload your MOF text file. For example, the command **mofcomp -N:root/wmi driver.mof** will try to recompile and reload any MOF data in the driver.mof file. Check to see what error messages Mofcomp generates in mofcomp.log. (Note that if your MOF file uses preprocessor directives such as **#define**, you will need to use the already-preprocessed MOF file, and not the original source file.

   Warning If this operation succeeds, it actually registers the new WMI class data with the system. You will need to delete these classes (by using Wbemtest, for example) to test if your driver's MOF data is being read correctly.
7. If the previous step succeeds, then the most likely problem is that the members of **WMIREGINFO**, such as **MofResourceName**, are specified incorrectly. Alternatively, the problem could be that your MOF file specifies a class derived from a base class that does not exist.
8. If the driver is using dynamic MOF data (see Implementing Dynamic MOF Data), check that the driver is receiving WMI IRP requests for the MSWmi_MofData_GUID GUID and that it is completing the IRP successfully and with no error logged.

Driver's WMI Properties or Methods Cannot Be Accessed

1. Use **wmimofck driver.bmf** to check if the binary MOF file format is correct. For more information, see Using wmimofck.exe.
2. Check the system event log for errors. For more information, see WMI IRPs and the System Event Log.
3. Check the WMI WDM Provider Log for errors.
4. Make sure the driver receives a WMI IRP whenever you use Wbemtest to query the driver's classes. If not, then check that the specified GUID in the MOF file matches the GUID the driver is expecting. Also check that the driver is receiving the WMI registration request, that it is succeeding, and the driver is registering the right GUIDs.
5. If the driver receives the IRP, ensure that the IRP is completed successfully, and that the driver is returning the right type of **WNODE_*XXX*** structure.
6. If Wbemtest returns an error, click the **More Information** button and check the **Description** property for a description of the error.

7. For methods, check that your driver supports handling the **IRP_MN_QUERY_ALL_DATA** and **IRP_MN_QUERY_SINGLE_INSTANCE** requests for the method's GUID. WMI will always perform one of those two requests before executing a method.

Driver's WMI Events Are Not Being Received

1. Check the system event log for errors. For example, if the driver specifies a static event name when calling **IoWMIWriteEvent** but the driver did not register any static event names, this would produce an entry in the system event log.

2. Check the WMI WDM provider log for errors.

3. If the driver is sending an event reference, the driver should receive an **IRP_MN_QUERY_SINGLE_INSTANCE** request immediately after sending the event reference. If the driver does not receive the IRP, the **WNODE_EVENT_REFERENCE** structure may have been malformed. If the driver receives the IRP, it should be completing it with status STATUS_SUCCESS.

4. If the driver uses **IoWMIWriteEvent** to send the event or event reference, make sure the event structure (either **WNODE_SINGLE_INSTANCE** or **WNODE_EVENT_REFERENCE**) is filled out correctly. In particular, if the event GUID is registered for static instance names, make sure that the correct instance index and provider ID are provided. If the event GUID is registered for dynamic instance names, make sure the instance name is included when the event is sent. If using the **WNODE_EVENT_REFERENCE** structure to specify the event, check that **Wnode.Guid** matches **TargetGuid**.

5. If the driver uses **WmiFireEvent** to send the event, make sure the correct value is passed for the *Guid* and *InstanceIndex* parameters.

Changes in Security Settings for WMI Requests Do Not Take Effect

- Unload and reload the WMI WDM Provider. For WMI data blocks registered with the WMIREG_FLAG_EXPENSIVE flag, the provider keeps a handle open to the data block as long as there are consumers for that block. The new security settings will not take effect until the provider closes the handle. Unloading and reloading the provider makes sure the handle has been closed. (For more information about the WMIREG_FLAG_EXPENSIVE flag, see **WMIREGGUID**.)

Using Nt and Zw Versions of the Native System Services Routines

6/25/2019 • 2 minutes to read • Edit Online

The Windows native operating system services API is implemented as a set of routines that run in kernel mode. These routines have names that begin with the prefix **Nt** or **Zw**. Kernel-mode drivers can call these routines directly. User-mode applications can access these routines by using system calls.

With a few exceptions, each native system services routine has two slightly different versions that have similar names but different prefixes. For example, calls to NtCreateFile and **ZwCreateFile** perform similar operations and are, in fact, serviced by the same kernel-mode system routine. For system calls from user mode, the **Nt** and **Zw** versions of a routine behave identically. For calls from a kernel-mode driver, the **Nt** and **Zw** versions of a routine differ in how they handle the parameter values that the caller passes to the routine.

A kernel-mode driver calls the **Zw** version of a native system services routine to inform the routine that the parameters come from a trusted, kernel-mode source. In this case, the routine assumes that it can safely use the parameters without first validating them. However, if the parameters might be from either a user-mode source or a kernel-mode source, the driver instead calls the **Nt** version of the routine, which determines, based on the history of the calling thread, whether the parameters originated in user mode or kernel mode. For more information about how the routine distinguishes user-mode parameters from kernel-mode parameters, see **PreviousMode**.

When a user-mode application calls the **Nt** or **Zw** version of a native system services routine, the routine always treats the parameters that it receives as values that come from a user-mode source that is not trusted. The routine thoroughly validates the parameter values before it uses the parameters. In particular, the routine probes any caller-supplied buffers to verify that the buffers are located in valid user-mode memory and are aligned properly.

Native system services routines make additional assumptions about the parameters that they receive. If a routine receives a pointer to a buffer that was allocated by a kernel-mode driver, the routine assumes that the buffer was allocated in system memory, not in user-mode memory. If the routine receives a handle that was opened by a user-mode application, the routine looks for the handle in the user-mode handle table, not in the kernel-mode handle table.

In a few instances, the meaning of a parameter value differs more significantly between calls from user mode and from kernel mode. For example, the **ZwNotifyChangeKey** routine (or its **NtNotifyChangeKey** counterpart) has a pair of input parameters, *ApcRoutine* and *ApcContext*, that mean different things, depending on whether the parameters are from a user-mode or kernel-mode source. For a call from user mode, *ApcRoutine* points to an APC routine and *ApcContext* points to a context value that the operating system supplies when it calls the APC routine. For a call from kernel mode, *ApcRoutine* points to a **WORK_QUEUE_ITEM** structure, and *ApcContext* specifies the type of work queue item that is described by the **WORK_QUEUE_ITEM** structure.

This section includes the following topics:

PreviousMode

6/25/2019 • 2 minutes to read • Edit Online

When a user-mode application calls the **Nt** or **Zw** version of a native system services routine, the system call mechanism traps the calling thread to kernel mode. To indicate that the parameter values originated in user mode, the trap handler for the system call sets the **PreviousMode** field in the thread object of the caller to **UserMode**. The native system services routine checks the **PreviousMode** field of the calling thread to determine whether the parameters are from a user-mode source.

If a kernel-mode driver calls a native system services routine and passes parameter values to the routine that are from a kernel-mode source, the driver must make sure that the **PreviousMode** field in the current thread object is set to **KernelMode**.

A kernel-mode driver can run in the context of an arbitrary thread, and the **PreviousMode** field of this thread might be set to **UserMode**. In this situation, a kernel-mode driver can call the **Zw** version of a native system services routine to inform the routine that the parameter values are from a trusted, kernel-mode source. The **Zw** call goes to a thin wrapper function that overrides the **PreviousMode** value in the current thread object. The wrapper function sets **PreviousMode** to **KernelMode** and calls the **Nt** version of the routine. On return from the **Nt** version of the routine, the wrapper function restores the original **PreviousMode** value of the thread object and returns.

A kernel-mode driver can directly call the **Nt** version of a native system services routine. When a kernel-mode driver processes an I/O request that can originate either in user mode or in kernel mode, the driver can call the **Nt** version of the routine so that the **PreviousMode** value of the current thread remains unaltered during the call. The **Nt*Xxx*** routine checks the calling thread's **PreviousMode** value to determine whether the parameter values are from a user-mode application or a kernel-mode component, and treats them accordingly.

An error can occur if a kernel-mode driver calls an **Nt*Xxx*** routine and the **PreviousMode** value in the current thread object does not accurately indicate whether the parameter values are from a user-mode or a kernel-mode source.

For example, assume that a kernel-mode driver is running in the context of an arbitrary thread, and that the **PreviousMode** value for this thread is set to **UserMode**. If the driver passes a kernel-mode file handle to the **NtClose** routine, this routine checks the **PreviousMode** value and decides that the handle must be a user-mode handle. When **NtClose** does not find the handle in the user-mode handle table, it returns the STATUS_INVALID_HANDLE error code. Meanwhile, the driver leaks the kernel-mode handle, which was never closed.

For another example, if the parameters for an **Nt*Xxx*** routine include an input or output buffer, and if **PreviousMode** = **UserMode**, the routine calls the **ProbeForRead** or **ProbeForWrite** routine to validate the buffer. If the buffer was allocated in system memory instead of in user-mode memory, the **ProbeFor*Xxx*** routine raises an exception, and the **Nt*Xxx*** routine returns the STATUS_ACCESS_VIOLATION error code.

If it is necessary, a driver can call the **ExGetPreviousMode** routine to get the **PreviousMode** value from the current thread object. Or, the driver can read the **RequestorMode** field from the **IRP** structure that describes the requested I/O operation. The **RequestorMode** field contains a copy of the **PreviousMode** value from the thread that requested the operation.

Libraries and Headers

6/25/2019 • 2 minutes to read • Edit Online

Kernel-mode drivers use the native system services routines by calling the **Nt** and **Zw** entry points in the Ntoskrnl.exe dynamic link library (DLL). This DLL contains the actual implementations of these routines. To access these entry points, a driver statically links to the Ntoskrnl.lib library, which is available in the Windows Driver Kit (WDK). The routines that are implemented in Ntoskrnl.lib are stubs that dynamically link to the entry points in Ntoskrnl.exe at run time.

The WDK documentation describes some, but not all, of the **Zw** entry points in Ntoskrnl.exe. For descriptions of the **Zw** routines that can be called by drivers, see ZwXxx Routines.

Most of the documented **Zw** routines are defined in the Wdm.h header file in the WDK, but a few are defined in other header files, such as Ntddk.h and Ntifs.h.

Typically, user-mode applications do not call the **Nt** and **Zw** routines. Instead, an application might call a Win32 routine, such as CreateFile, which then calls a native system services routine, such as NtCreateFile or **ZwCreateFile**, to perform the requested operation. However, a user-mode application might directly call an **Nt** or **Zw** routine to perform an operation that is not supported by the Win32 routines.

User-mode applications use the native system services routines by calling the entry points in the Ntdll.dll dynamic link library. These entry points convert calls to **Nt** and **Zw** routines into system calls that are trapped to kernel mode. To access these entry points, a user-mode application statically links to the Ntdll.lib library, which is available in the WDK. The routines that are implemented in Ntdll.lib are stubs that dynamically link to the entry points in Ntdll.dll at run time.

The Windows SDK documentation describes some, but not all, of the **Nt** entry points in Ntdll.lib. Most of the documented **Nt** routines are defined in the Winternl.h header file in the Windows SDK. This documentation makes little mention of the **Zw** entry points, and no header files in the Windows SDK contain definitions of **Zw** routines.

With a couple of minor exceptions, each entry point in Ntdll.dll for an **Nt** routine has a matching entry point for a **Zw** routine. The documentation for the WDK and Windows SDK recommends that application developers avoid calling undocumented **Nt** entry points, and warns that the **Zw** entry points might disappear from Ntdll.dll in a future version of Windows. Application developers who call the **Zw** routines from user mode should be prepared for this occurrence.

For descriptions of the **Nt** routines that can be called by applications, see Winternl, Files, and Miscellaneous Low-Level Client Support. Some reference pages for **Nt** routines in the Windows SDK documentation label the routines as "deprecated" and advise readers to use the equivalent Win32 routines instead of the deprecated **Nt** routines.

A user-mode application cannot call the entry points in Ntoskrnl.exe, and a kernel-mode driver cannot call the entry points in Ntdll.dll.

What Does the Zw Prefix Mean?

6/25/2019 • 2 minutes to read • Edit Online

The Windows native system services routines have names that begin with the prefixes **Nt** and **Zw**. The **Nt** prefix is an abbreviation of Windows NT, but the **Zw** prefix has no meaning. **Zw** was selected partly to avoid potential naming conflicts with other APIs, and partly to avoid using any potentially useful two-letter prefixes that might be needed in the future.

Many of the Windows driver support routines have names that begin with two- or three-letter prefixes. These prefixes indicate which kernel-mode system components implement the routines. The following table contains some examples.

PREFIX	KERNEL COMPONENT	EXAMPLE ROUTINE
Cm	Configuration manager	**CmRegisterCallbackEx**
Ex	Executive	**ExAllocatePool**
Hal	Hardware abstraction layer	**HalGetAdapter**
Io	I/O manager	**IoAllocateIrp**
Ke	Kernel core	**KeSetEvent**
Mm	Memory manager	**MmUnlockPages**
Ob	Object manager	**ObReferenceObject**
Po	Power manager	**PoSetPowerState**
Tm	Transaction manager	**TmCommitTransaction**
Nt and **Zw**	Native system services	NtCreateFile and **ZwCreateFile**

Specifying Access Rights

10/7/2019 • 2 minutes to read • Edit Online

The ACCESS_MASK type is a bitmask that specifies a set of access rights in the access mask of an access control entry.

```
typedef ULONG  ACCESS_MASK;
```

The following standard specific access rights apply to all types of executive objects.

FLAG	DESCRIPTION
DELETE	The caller can delete the object.
READ_CONTROL	The caller can read the access control list (ACL) and ownership information for the file.
SYNCHRONIZE	The caller can perform a wait operation on the object. (For example, the object can be passed to **KeWaitForMultipleObjects**.)
WRITE_DAC	The caller can change the discretionary access control list (DACL) information for the object.
WRITE_OWNER	The caller can change the ownership information for the file.

Note that normally only DELETE and SYNCHRONIZE are of interest to driver writers.

You can also specify the following generic access rights. These also apply to all types of executive objects. The meaning of each generic access right is specific to that type of object.

FLAG	DESCRIPTION
GENERIC_READ	The caller can perform normal read operations on the object.
GENERIC_WRITE	The caller can perform normal write operations on the object.
GENERIC_EXECUTE	The caller can execute the object. (Note this generally only makes sense for certain kinds of objects, such as file objects and section objects.)

FLAG	DESCRIPTION
GENERIC_ALL	The caller can perform all normal operations on the object.

The following combinations of standard specific access rights are also defined. These are not normally used directly, but are used as templates to define other bitmasks. (For example, when you specify GENERIC_READ for a file object, the system maps this to the FILE_GENERIC_READ bitmask of specific access rights. FILE_GENERIC_READ is defined in terms of STANDARD_RIGHTS_READ.)

BITMASK	DESCRIPTION
STANDARD_RIGHTS_READ	Standard specific rights that correspond to GENERIC_READ
STANDARD_RIGHTS_WRITE	Standard specific rights that correspond to GENERIC_WRITE
STANDARD_RIGHTS_EXECUTE	Standard specific rights that correspond to GENERIC_EXECUTE
STANDARD_RIGHTS_REQUIRED	Standard specific rights that correspond to GENERIC_ALL. This includes DELETE, but not SYNCHRONIZE.
STANDARD_RIGHTS_ALL	All standard access rights.

Each type of object can have its own additional access rights. For a description of the access rights that are applicable to a file, directory, or device, see **ZwCreateFile**. For a description of the access rights that are applicable to an object manager directory, see **ZwCreateDirectoryObject**. For a description of the access rights that are applicable to a registry key, see **ZwCreateKey**. For a description of the access rights that are applicable to a section object, see **ZwOpenSection**. For a description of the access rights that are applicable to a WMI data block, see **IoWMIOpenBlock**.

For more information about access rights, see the following topics in the Microsoft Windows SDK documentation:

- Access Rights and Access Masks
- ACCESS_MASK

Wdm.h (include Wdm.h, Ntddk.h, or Ntifs.h)

Related topics

IoWMIOpenBlock
ZwCreateDirectoryObject
ZwCreateFile
ZwCreateKey
ZwOpenSection

NtXxx Routines

6/25/2019 • 2 minutes to read • Edit Online

This section describes the **Nt*Xxx*** versions of the Windows Native System Services routines. Most native system services routines have two versions, one of which has a name begins with the prefix **Nt**; the other version has a name that begins with the prefix **Zw**. For example, calls to **NtCreateFile** and **ZwCreateFile** perform similar operations and are, in fact, serviced by the same kernel-mode system routine.

For calls from kernel-mode drivers, the **Nt*Xxx*** and **Zw*Xxx*** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt*Xxx*** and **Zw*Xxx*** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

The following table summarizes the **Nt*Xxx*** and **Zw*Xxx*** versions of the routines:

NTXXX	ZWXXX
NtAllocateLocallyUniqueId	**ZwAllocateLocallyUniqueId**
NtAllocateVirtualMemory	**ZwAllocateVirtualMemory**
NtClose	**ZwClose**
NtCommitComplete	**ZwCommitComplete**
NtCommitEnlistment	**ZwCommitEnlistment**
NtCommitTransaction	**ZwCommitTransaction**
NtCreateDirectoryObject	**ZwCreateDirectoryObject**
NtCreateEnlistment	**ZwCreateEnlistment**
NtCreateEvent	**ZwCreateEvent**
NtCreateFile	**ZwCreateFile**
NtCreateKey	**ZwCreateKey**
NtCreateResourceManager	**ZwCreateResourceManager**

NTXXX	ZWXXX
NtCreateSection	**ZwCreateSection**
NtCreateTransaction	**ZwCreateTransaction**
NtCreateTransactionManager	**ZwCreateTransactionManager**
NtCurrentProcess	**ZwCurrentProcess**
NtCurrentThread	**ZwCurrentThread**
NtDeleteFile	**ZwDeleteFile**
NtDeleteKey	**ZwDeleteKey**
NtDeleteValueKey	**ZwDeleteValueKey**
NtDeviceIoControlFile	**ZwDeviceIoControlFile**
NtDuplicateObject	**ZwDuplicateObject**
NtDuplicateToken	**ZwDuplicateToken**
NtEnumerateKey	**ZwEnumerateKey**
NtEnumerateTransactionObject	**ZwEnumerateTransactionObject**
NtEnumerateValueKey	**ZwEnumerateValueKey**
NtFlushBuffersFile	**ZwFlushBuffersFile**
NtFlushBuffersFileEx	**ZwFlushBuffersFileEx**
NtFlushKey	**ZwFlushKey**
NtFlushVirtualMemory	**ZwFlushVirtualMemory**
NtFreeVirtualMemory	**ZwFreeVirtualMemory**

NTXXX	ZWXXX
NtFsControlFile	**ZwFsControlFile**
NtGetNotificationResourceManager	**ZwGetNotificationResourceManager**
NtLoadDriver	**ZwLoadDriver**
NtLockFile	**ZwLockFile**
NtMakeTemporaryObject	**ZwMakeTemporaryObject**
NtMapViewOfSection	**ZwMapViewOfSection**
NtNotifyChangeKey	**ZwNotifyChangeKey**
NtOpenDirectoryObject	**ZwOpenDirectoryObject**
NtOpenEnlistment	**ZwOpenEnlistment**
NtOpenEvent	**ZwOpenEvent**
NtOpenFile	**ZwOpenFile**
NtOpenKey	**ZwOpenKey**
NtOpenProcess	**ZwOpenProcess**
NtOpenProcessTokenEx	**ZwOpenProcessTokenEx**
NtOpenResourceManager	**ZwOpenResourceManager**
NtOpenSection	**ZwOpenSection**
NtOpenSymbolicLinkObject	**ZwOpenSymbolicLinkObject**
NtOpenThreadTokenEx	**ZwOpenThreadTokenEx**
NtOpenTransaction	**ZwOpenTransaction**

NTXXX	ZWXXX
NtOpenTransactionManager	**ZwOpenTransactionManager**
NtPowerInformation	**ZwPowerInformation**
NtPrepareComplete	**ZwPrepareComplete**
NtPrepareEnlistment	**ZwPrepareEnlistment**
NtPrePrepareComplete	**ZwPrePrepareComplete**
NtPrePrepareEnlistment	**ZwPrePrepareEnlistment**
NtQueryDirectoryFile	**ZwQueryDirectoryFile**
NtQueryFullAttributesFile	**ZwQueryFullAttributesFile**
NtQueryInformationEnlistment	**ZwQueryInformationEnlistment**
NtQueryInformationFile	**ZwQueryInformationFile**
NtQueryInformationResourceManager	**ZwQueryInformationResourceManager**
NtQueryInformationToken	**ZwQueryInformationToken**
NtQueryInformationTransaction	**ZwQueryInformationTransaction**
NtQueryInformationTransactionManager	**ZwQueryInformationTransactionManager**
NtQueryKey	**ZwQueryKey**
NtQueryObject	**ZwQueryObject**
NtQueryQuotaInformationFile	**ZwQueryQuotaInformationFile**
NtQuerySecurityObject	**ZwQuerySecurityObject**
NtQuerySecurityObject	**ZwQuerySymbolicLinkObject**

NTXXX	ZWXXX
NtQueryValueKey	**ZwQueryValueKey**
NtQueryVirtualMemory	**ZwQueryVirtualMemory**
NtQueryVolumeInformationFile	**ZwQueryVolumeInformationFile**
NtReadFile	**ZwReadFile**
NtReadOnlyEnlistment	**ZwReadOnlyEnlistment**
NtReadOnlyEnlistment	**ZwRecoverEnlistment**
NtRecoverResourceManager	**ZwRecoverResourceManager**
NtRecoverTransactionManager	**ZwRecoverTransactionManager**
NtRollbackComplete	**ZwRollbackComplete**
NtRollbackEnlistment	**ZwRollbackEnlistment**
NtRollbackTransaction	**ZwRollbackTransaction**
NtRollforwardTransactionManager	**ZwRollforwardTransactionManager**
NtSetEvent	**ZwSetEvent**
NtSetInformationEnlistment	**ZwSetInformationEnlistment**
NtSetInformationFile	**ZwSetInformationFile**
NtSetInformationResourceManager	**ZwSetInformationResourceManager**
NtSetInformationThread	**ZwSetInformationThread**
NtSetInformationToken	**ZwSetInformationToken**
NtSetInformationTransaction	**ZwSetInformationTransaction**

NTXXX	ZWXXX
NtSetQuotaInformationFile	**ZwSetQuotaInformationFile**
NtSetSecurityObject	**ZwSetSecurityObject**
NtSetValueKey	**ZwSetValueKey**
NtSetVolumeInformationFile	**ZwSetVolumeInformationFile**
NtSinglePhaseReject	**ZwSinglePhaseReject**
NtTerminateProcess	**ZwTerminateProcess**
NtUnloadDriver	**ZwUnloadDriver**
NtUnlockFile	**ZwUnlockFile**
NtUnmapViewOfSection	**ZwUnmapViewOfSection**
NtWaitForSingleObject	**ZwWaitForSingleObject**
NtWriteFile	**ZwWriteFile**

Introduction to Kernel Dispatcher Objects

6/25/2019 • 3 minutes to read • Edit Online

The kernel defines a set of object types called *kernel dispatcher objects*, or just *dispatcher objects*. Dispatcher objects include timer objects, event objects, semaphore objects, mutex objects, and thread objects.

Drivers can use dispatcher objects as synchronization mechanisms within a nonarbitrary thread context while executing at IRQL PASSIVE_LEVEL.

Dispatcher Object States

Every kernel-defined dispatcher object type has a state that is either set to Signaled or set to Not-Signaled.

A group of threads can synchronize their operations if one or more threads call **KeWaitForSingleObject**, **KeWaitForMutexObject**, or **KeWaitForMultipleObjects**. These functions take dispatcher object pointers as input and wait until another routine or thread sets one or more dispatcher objects to the Signaled state.

When a thread calls the **KeWaitForSingleObject** to wait for a dispatcher object (or **KeWaitForMutexObject** for a mutex), the thread is put into a *wait* state until the dispatcher object is set to the Signaled state. A thread can call **KeWaitForMultipleObjects** to wait either for any, or for all, of a set of dispatcher objects to be set to Signaled.

Whenever a dispatcher object is set to the Signaled state, the kernel changes the state of any thread waiting for that object to *ready*. (Synchronization timers and synchronization events are exceptions to this rule; when a synchronization event or timer is signaled, only one waiting thread is set to the ready state. For more information, see Timer Objects and DPCs and Event Objects.) A thread in the ready state will be scheduled to run according to its current run-time thread priority and the current availability of processors for any thread with that priority.

When Can Drivers Wait for Dispatcher Objects?

In general, drivers can wait for dispatcher objects to be set only if at least one of the following circumstances is true:

- The driver is executing in a nonarbitrary thread context.

  That is, you can identify the thread that will enter a wait state. In practice, the only driver routines that execute in a nonarbitrary thread context are the **DriverEntry**, *AddDevice*, *Reinitialize*, and *Unload* routines of any driver, plus the dispatch routines of highest-level drivers. All these routines are called directly by the system.

- The driver is performing a completely synchronous I/O request.

  That is, no driver queues any operations while handling the I/O request, and no driver returns until the driver below it has finished handling the request.

Additionally, a driver cannot enter a wait state if it is executing at or above IRQL = DISPATCH_LEVEL.

Based on these limitations, you must use the following rules:

- The **DriverEntry**, *AddDevice*, *Reinitialize*, and *Unload* routines of any driver can wait for dispatcher objects.
- The dispatch routines of a highest-level driver can wait for dispatcher objects.
- The dispatch routines of lower-level drivers can wait for dispatch objects, if the I/O operation is synchronous, such as create, flush, shutdown, and close operations, some device I/O control operations, and some PnP and power operations.
- The dispatch routines of lower-level drivers cannot wait for a dispatcher object for the completion of asynchronous I/O operations.

- A driver routine that is executing at or above IRQL DISPATCH_LEVEL must not wait for a dispatcher object to be set to the Signaled state.
- A driver must not attempt to wait for a dispatcher object to be set to the Signaled state for the completion of a transfer operation to or from a paging device.
- Driver dispatch routines servicing read/write requests generally cannot wait for a dispatcher object to be set to the Signaled state.
- A dispatch routine for a device I/O control request can wait for a dispatcher object to be set to the Signaled state only if the transfer type for the I/O control code is METHOD_BUFFERED.
- SCSI miniport drivers should not use kernel dispatcher objects. SCSI miniport drivers should call only SCSI Port Library Routines.

Every other standard driver routine executes in an arbitrary thread context: that of whatever thread happens to be current when the driver routine is called to process a queued operation or to handle a device interrupt. Moreover, most standard driver routines are run at a raised IRQL, either at DISPATCH_LEVEL, or for device drivers, at DIRQL.

If necessary, a driver can create a device-dedicated thread, which can wait for the driver's other routines (except an ISR or *SynchCritSection* routine) to set a dispatcher object to the Signaled state and reset to the Not-Signaled state.

As a general guideline, if you expect that your new device driver will often need to stall for longer than 50 microseconds while it waits for device-state changes during I/O operations, consider implementing a driver with a device-dedicated thread. If the device driver is also a highest-level driver, consider using system worker threads and implementing one or more worker-thread callback routines. See **PsCreateSystemThread** and Managing Interlocked Queues with a Driver-Created Thread.

Timer Objects

6/25/2019 • 2 minutes to read • Edit Online

Any driver can use a timer object within a nonarbitrary thread context to time-out operations in the driver's other routines, or to schedule operations to be performed periodically. Starting with Windows 2000, timer objects based on the **KTIMER** structure are available to use with **KeSetTimer** and the other **Ke*Xxx*Timer** routines. Starting with Windows 8.1, timer objects based on the **EX_TIMER** structure are available to use with **ExSetTimer** and the other **Ex*Xxx*Timer** routines. Timer objects based on the **KTIMER** and **EX_TIMER** structures are kernel dispatcher objects that are signaled when a timer expires. Timer expiration can be periodic or one-shot (nonperiodic).

This section contains the following topics:

- KeXxxTimer Routines, KTIMER Objects, and DPCs
- ExXxxTimer Routines and EX_TIMER Objects

KeXxxTimer Routines, KTIMER Objects, and DPCs

6/25/2019 • 2 minutes to read • Edit Online

Starting with Windows 2000, a set of **Ke*Xxx*Timer** routines is available to manage timers. These routines use timer objects that are based on the **KTIMER** structure. To create a timer object, a driver first allocates storage for a **KTIMER** structure. Then the driver calls a routine such as **KeInitializeTimer** or **KeInitializeTimerEx** to initialize this structure.

A timer can be set to expire just once, or to expire repeatedly after a given interval. **KeSetTimer** always sets a timer that will expire just once. **KeSetTimerEx** accepts an optional *Period* parameter, which specifies a recurring timer interval.

An optional *CustomTimerDpc* routine (a type of deferred procedure call) can be associated with either a notification timer or a synchronization timer. This routine executes when the specified time interval expires. For more information, see Using Timer Objects.

A timer can be a *notification timer* or a *synchronization timer*.

- When a notification timer is signaled, all waiting threads have their wait satisfied. The state of the timer remains signaled until it is explicitly reset.
- When a synchronization timer expires, its state is set to Signaled until a single waiting thread is released. Then the timer is reset to the Not-Signaled state.

KeInitializeTimer always creates notification timers. **KeInitializeTimerEx** accepts a *Type* parameter, which can be **NotificationTimer** or **SynchronizationTimer**.

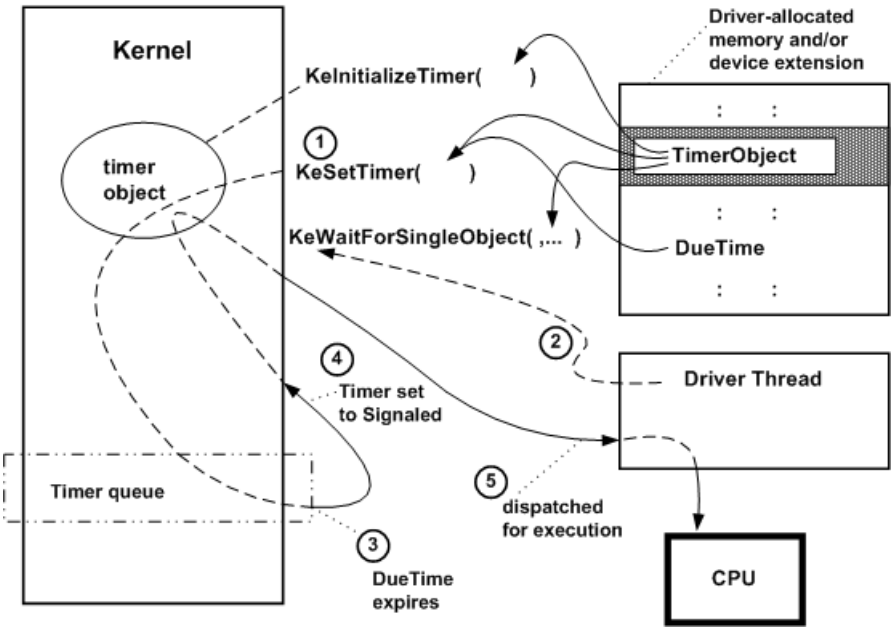
The following topics provide more information about timer objects and DPCs:

Using Timer Objects Timer Accuracy Registering and Queuing a CustomTimerDpc Routine Providing CustomTimerDpc Context Information Using a CustomTimerDpc Routine

Using Timer Objects

6/25/2019 • 3 minutes to read • Edit Online

The following figure illustrates the use of a notification timer to set up a timeout interval for an operation and then wait while other driver routines process an I/O request.



As the previous figure shows, a driver must provide storage for the timer object, which must be initialized by a call to **KeInitializeTimer** with a pointer to this storage. A driver typically makes this call from its *AddDevice* routine.

Within the context of a particular thread, such as a driver-created thread or a thread requesting a synchronous I/O operation, the driver can wait for its timer object as shown in the previous figure:

1. The thread calls **KeSetTimer** with a pointer to the timer object and a given *DueTime* value, expressed in units of 100 nanoseconds. A positive value for *DueTime* specifies an absolute time at which the timer object should be removed from the kernel's timer queue and set to the Signaled state. A negative value for *DueTime* specifies an interval relative to the current system time.

   Note that the thread (or driver routine running in a system thread) passes a **NULL** pointer for the DPC object (shown previously in the figure illustrating using timer and DPC objects for a CustomTimerDpc routine) when it calls **KeSetTimer** if it waits on the timer object instead of queuing a *CustomTimerDpc* routine.

2. The thread calls **KeWaitForSingleObject** with a pointer to the timer object, which puts the thread into a wait state while the timer object is in the kernel's timer queue.

3. The given *DueTime* expires.

4. The kernel dequeues the timer object, sets it to the Signaled state, and changes the thread's state from waiting to ready.

5. The kernel dispatches the thread for execution as soon as a processor is available: that is, no other thread with a higher priority is currently in the ready state and there are no kernel-mode routines to be run at a higher IRQL.

Driver routines that run at IRQL >= DISPATCH_LEVEL can time out requests by using a timer object with an associated DPC object to queue a driver-supplied *CustomTimerDpc* routine. Only driver routines that run within a

nonarbitrary thread context can wait for a nonzero interval on a timer object, as shown in the previous figure.

Like every other thread, a driver-created thread is represented by a kernel thread object, which is also a dispatcher object. Consequently, a driver need not have its driver-created thread use a timer object to voluntarily put itself into a wait state for a given interval. Instead, the thread can call **KeDelayExecutionThread** with a caller-supplied interval. For more information about this technique, see Polling a Device.

DriverEntry, *Reinitialize*, and *Unload* routines also run in a system thread context, so drivers can call **KeWaitForSingleObject** with a driver-initialized timer object or **KeDelayExecutionThread** while they are initializing or unloading. A device driver can call **KeStallExecutionProcessor** for a very short interval (preferably something less than 50 microseconds) if it must wait for the device to update state during its initialization.

However, higher-level drivers generally use another synchronization mechanism in their **DriverEntry** and *Reinitialize* routines instead of using a timer object. Higher-level drivers should always be designed to layer themselves over any lower-level driver of a particular type or types of device. Therefore, a higher-level driver tends to become slow to load if it waits on a timer object or calls **KeDelayExecutionThread** because such a driver must wait for an interval long enough to accommodate the slowest possible device supporting it. Note also that a "safe" but minimum interval for such a wait is very difficult to determine.

Similarly, PnP drivers should not wait for other actions to occur, but instead should use the PnP manager's notification mechanism.

Timer Accuracy

6/25/2019 • 2 minutes to read • Edit Online

A system timer routine typically enables the caller to specify either an absolute or a relative expiration time for a timer. For example, see **KeWaitForSingleObject**, **KeSetTimer**, or **KeDelayExecutionThread**. The accuracy with which the operating system can measure expiration times is limited by the granularity of the system clock.

The system time is updated on every tick of the system clock, and is accurate only to the latest tick. If the caller specifies an absolute expiration time, the expiration of the timer is detected during processing of the first system clock tick that occurs after the specified time. Thus, the timer can expire as much as one system clock period later than the specified absolute expiration time. If a timer interval, or relative expiration time, is instead specified, the expiration can occur up to a period earlier than or a period later than the specified time, depending on where exactly the start and end times of this interval fall between system clock ticks. Regardless of whether an absolute or a relative time is specified, the timer expiration might not be detected until even later if interrupt processing for the system clock is delayed by interrupt processing for other devices.

When the caller specifies a relative expiration time, the timer routine adds the current system clock time to the specified relative expiration time to calculate the absolute expiration time to use for the timer. Because the system time is accurate only to the latest tick of the system clock, the calculated expiration time can be up to a system clock period earlier than the expiration time expected by the caller. If a specified relative expiration time is close to or smaller than the system clock period, the timer might expire immediately, with no delay.

A possible way to more accurately support shorter expiration times is to decrease the time between system clock ticks, but doing so is likely to increase power consumption. In addition, reducing the system clock period might not reliably achieve a finer system clock granularity unless interrupt processing for the other devices in the platform can be guaranteed not to delay the processing of system clock interrupts.

Starting with Windows 8, **KeDelayExecutionThread** uses a more precise technique to calculate the absolute expiration time from a caller-specified relative expiration time. First, to obtain a more precise estimate of the current system time, the routine uses the system performance counter to measure the time elapsed since the last system clock tick. Next, the routine adds this more precise estimate of the system time to the relative expiration time to calculate the absolute expiration time. The absolute expiration time calculated by this technique is accurate to within a microsecond. As a result, the timer will not expire before the specified relative expiration time elapses. The timer can still expire up to a system clock period later than the specified time, and might expire even later if processing of the system clock interrupt is delayed by interrupt processing for other devices.

If the system time changes before a timer expires, a relative timer is not affected but the system adjusts each absolute timer. A relative timer always expires after the specified number of time units elapse, regardless of the absolute system time. An absolute timer expires at a specific system time, so a change in the system time changes the wait duration of an absolute timer.

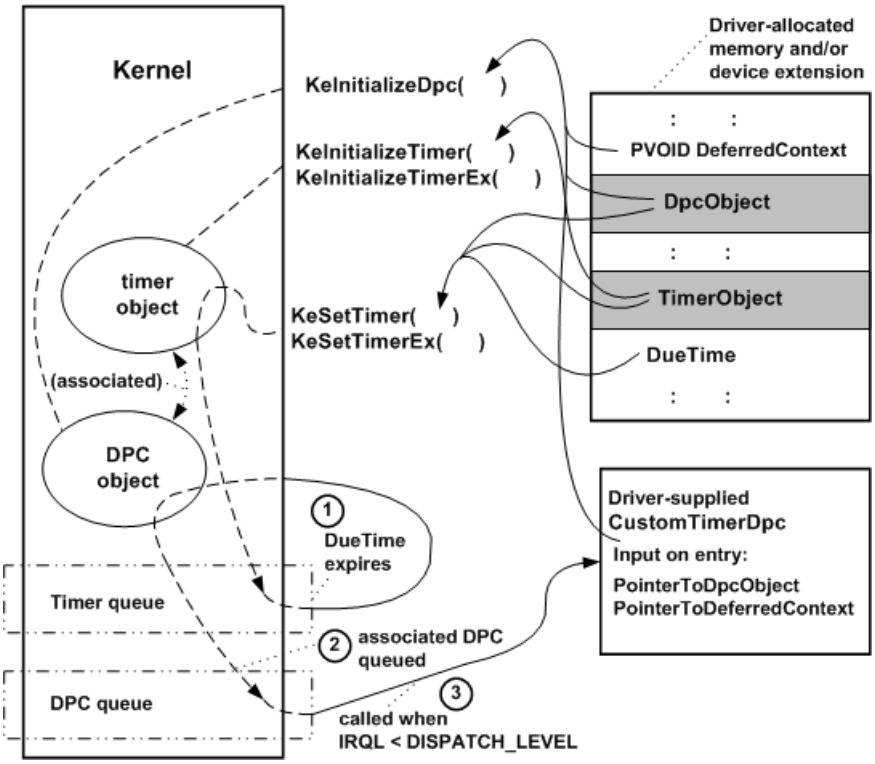
Registering and Queuing a CustomTimerDpc Routine

6/25/2019 • 2 minutes to read • Edit Online

A driver can register a *CustomTimerDpc* routine by calling the following routines, usually from its *AddDevice* routine:

1. **KeInitializeDpc** to register its routine
2. **KeInitializeTimer** or **KeInitializeTimerEx** to set up a timer object

Subsequently, the driver can call **KeSetTimer** or **KeSetTimerEx** to specify an expiration time and to add the timer object to the system's timer queue. When the expiration time is reached, the system dequeues the timer object and calls the *CustomTimerDpc* routine. The following figure illustrates these calls.



As the previous figure shows, the driver must supply storage for both a DPC object and a timer object. Most drivers provide the storage for these objects in a device extension or in other driver-allocated, resident memory.

In the call to **KeSetTimer**, the driver passes pointers to the *Dpc* and *Timer* objects, along with a *DueTime* expressed in units of 100 nanoseconds, as shown in the previous figure. A positive value for *DueTime* specifies an *absolute expiration time* (since January 1, 1601) at which the *CustomTimerDpc* routine should be called. A negative value for *DueTime* specifies a *relative expiration time*.

Because an absolute timer expires at a specific system time, the wait duration of an absolute timer is not affected if the system time changes before the timer expires. On the other hand, a relative timer always expires after the specified number of time units elapses, regardless of changes to the absolute system time.

To invoke a *CustomTimerDpc* routine repeatedly, use **KeSetTimerEx** to set the timer and specify a recurring interval in the *Period* parameter. **KeSetTimerEx** is just like **KeSetTimer** except for this additional parameter.

As shown in the previous figure, the call to **KeSetTimer** or **KeSetTimerEx** queues the timer object for a specified interval as follows:

1. When the *DueTime* expires, the timer object is dequeued and set to the Signaled state.

2. If every processor in the machine is currently running code at an IRQL greater than or equal to DISPATCH_LEVEL, the DPC object associated with the timer object is put in a DPC queue. Otherwise, the *CustomTimerDpc* routine is called.

3. If the DPC object was already in the queue when the *DueTime* interval expired, the *CustomTimerDpc* routine is called as soon as the IRQL on any processor in the machine falls below DISPATCH_LEVEL.

   Note The *CustomTimerDpc* routine, like all DPC routines, is called at IRQL = DISPATCH_LEVEL. While a DPC routine runs, all threads are prevented from running on the same processor. Driver developers should carefully design their *CustomTimerDpc* routines to run for as brief a time as possible.

The smallest time interval that can be specified to **KeSetTimer** and **KeSetTimerEx** is approximately ten milliseconds, so a driver can use a *CustomTimerDpc* routine when timing smaller intervals than an *IoTimer* routine, which is run once per second, can handle.

Only one instantiation of a particular timer object can be queued at any moment. Calling **KeSetTimer** or **KeSetTimerEx** again with the same *Timer* object pointer cancels the queued timer object and resets it.

Setting up a *CustomTimerDpc* routine is exactly like setting up a *CustomDpc* routine, with an additional step to initialize the timer object. In fact, their prototypes are identical, but *CustomTimerDpc* routine cannot use the two *SystemArgument* pointers declared in its prototype.

Providing CustomTimerDpc Context Information

6/25/2019 • 2 minutes to read • Edit Online

The *DeferredContext* pointer passed to **KeInitializeDpc** points to a context area where other driver routines, and the *CustomTimerDpc* routine itself, can maintain state. The kernel passes the *DeferredContext* pointer in every call to the DPC routine.

Unlike an *IoTimer* routine, a *CustomTimerDpc* has no particular associations with a driver-created device object. However, a driver can associate a *CustomTimerDpc* routine with a driver-created device object by including a pointer to the device object in its context area.

The context area must be in resident, driver-allocated memory. Usually, this context area is in a device extension, but it can also be in nonpaged pool. If the driver uses a controller object, it can be in a controller extension. The contents of the context area are driver-determined.

If a *CustomTimerDpc* routine shares context information with the driver's ISR, the *CustomTimerDpc* routine must use **KeSynchronizeExecution** to call a *SynchCritSection* routine that accesses the shared context. For more information, see Using Critical Sections.

If the *CustomTimerDpc* shares the context information with other non-ISR driver routines, the area at *DeferredContext* must be protected by an executive spin lock. For more information, see Spin Locks.

Using a CustomTimerDpc Routine

6/25/2019 • 3 minutes to read • Edit Online

To disable a previously set timer object, a driver calls **KeCancelTimer**. This routine removes the timer object from the system's timer queue. Generally, the timer object is not set to the signaled state and the *CustomTimerDpc* routine is not queued for execution. However, if the timer is about to expire when **KeCancelTimer** is called, expiration might occur before **KeCancelTimer** has a chance to access the time queue, in which case signaling and DPC queuing will occur.

Recalling **KeSetTimer** or **KeSetTimerEx**, with previously specified *Timer* and *Dpc* pointers, before the previously specified interval expires, has the following effects:

- The kernel removes the timer object from the timer queue, without setting the object to the signaled state or queuing the *CustomTimerDpc* routine.
- The kernel reinserts the timer object in the timer queue, using the new *DueTime* value.

Using the same timer object for different purposes can cause race conditions or serious driver errors. For example, assume that a driver specifies a single timer object both to set up a call to a *CustomTimerDpc* routine and to set up waits in a driver-dedicated thread. Whenever the driver-dedicated thread calls **KeSetTimer**, **KeSetTimerEx**, or **KeCancelTimer** for the common timer object, the thread would cancel calls to the *CustomTimerDpc* routine, if the timer object was already queued for a *CustomTimerDpc* call.

If a driver has *CustomTimerDpc* routines, and also waits on timer objects in a nonarbitrary thread context, it should:

- Never use a thread-context-sensitive timer object in a nonarbitrary thread context, or vice versa.
- Allocate a separate timer object for each *CustomTimerDpc* routine. Each set of driver threads or driver routines that are called in a nonarbitrary thread context should have its own set of "waitable" timer objects.

If you use a *CustomTimerDpc* routine, choose carefully the interval the driver passes in calls to **KeSetTimer** or **KeSetTimerEx**. In addition, consider all possible effects of a call to **KeCancelTimer** with the same timer object from any driver routine that makes this call, particularly on SMP platforms.

Keep in mind the following fact about *CustomTimerDpc* routines:

Only one instantiation of a DPC object representing a particular DPC routine can be queued for execution at any given moment.

If a second driver routine calls **KeSetTimer** or **KeSetTimerEx** to run the same *CustomTimerDpc* routine before the interval specified by the first caller expires, the *CustomTimerDpc* routine is run only after the interval specified by the second caller expires. In these circumstances, the *CustomTimerDpc* does none of the work for which the first routine called **KeSetTimer** or **KeSetTimerEx**.

For drivers that have *CustomTimerDpc* routines and use periodic timers:

A driver cannot deallocate a periodic timer from a DPC routine. Drivers can deallocate only nonperiodic timers from a DPC routine.

Consider the following a design guideline for drivers that have both *CustomDpc* and *CustomTimerDpc* routines:

To prevent race conditions, never pass the same *Dpc* pointer to **KeSetTimer** or **KeSetTimerEx** and **KeInsertQueueDpc**.

In other words, suppose a driver's *StartIo* routine calls **KeSetTimer** or **KeSetTimerEx** to queue a *CustomTimerDpc* routine, and the driver's ISR calls **KeInsertQueueDpc** simultaneously from another processor with the same *Dpc* pointer. That DPC routine will be run when IRQL on a processor falls below DISPATCH_LEVEL or the timer interval expires, whichever comes first. Whichever does come first, some essential work for the *StartIo* or ISR would simply be dropped by the DPC routine.

In addition, a DPC used by two standard driver routines with very different functionality would have poorer performance characteristics than separate *CustomTimerDpc* and *CustomDpc* routines. The DPC would have to determine which operations to carry out, depending on the conditions that caused the *StartIo* routine or ISR to queue it. Testing for these conditions in the DPC would use additional CPU cycles.

ExXxxTimer Routines and EX_TIMER Objects

6/25/2019 • 2 minutes to read • Edit Online

Starting with Windows 8.1, a comprehensive set of **Ex*Xxx*Timer** routines is available to manage timers. These routines use timer objects that are based on the **EX_TIMER** structure. The **Ex*Xxx*Timer** routines are replacements for the **Ke*Xxx*Timer** routines, which are available starting with Windows 2000. Drivers intended to run only on Windows 8.1 and later versions of Windows can use the **Ex*Xxx*Timer** routines instead of the **Ke*Xxx*Timer** routines. Windows 8.1 and later versions of Windows continue to support the **Ke*Xxx*Timer** routines.

The **Ex*Xxx*Timer** routines have all the important capabilities that are provided by the **Ke*Xxx*Timer** routines. In addition, the **Ex*Xxx*Timer** routines support two timer types, *high-resolution timers* and *no-wake timers*, that are not supported by the **Ke*Xxx*Timer** routines. High-resolution timers are timers whose expiration times can be specified with higher accuracy than those of timers whose accuracy is limited by the default resolution of the system clock. No-wake timers are timers that avoid unnecessarily waking processors from low-power states. For more information, see the following topics:

High-Resolution Timers No-Wake Timers Starting with Windows 8.1, the following **Ex*Xxx*Timer** routines are available:

ExAllocateTimer ExSetTimer ExCancelTimer ExDeleteTimer The **ExSetTimer** routine can be used instead of the **KeSetTimer** or **KeSetTimerEx** routine. The **ExCancelTimer** routine can be used instead of the **KeCancelTimer** routine.

The **ExAllocateTimer** and **ExDeleteTimer** routines have no direct **Ke*Xxx*Timer** counterparts. These two routines allocate and free a timer object. This timer object is a system-allocated **EX_TIMER** structure whose members are opaque to drivers. In contrast, the timer object used by the **Ke*Xxx*Timer** routines is a driver-allocated **KTIMER** structure. The driver calls the **KeInitializeTimer** or **KeInitializeTimerEx** routine to initialize this object. **ExAllocateTimer** initializes the timer objects that it allocates. For more information about **ExDeleteTimer**, see Deleting a System-Allocated Timer Object.

EX_TIMER and **KTIMER** structures are waitable objects. After a driver calls **ExSetTimer**, **KeSetTimer**, or **KeSetTimerEx** to set a timer, the driver can call a routine such as **KeWaitForSingleObject** or **KeWaitForMultipleObjects** to wait for the timer to expire. The timer object is signaled when the timer expires. As an option, a driver can supply a pointer to a driver-implemented *ExTimerCallback* or *CustomTimerDpc* callback routine that the operating system calls after the timer expires.

The **Ke*Xxx*Timer** routines have two capabilities that are not provided by the **Ex*Xxx*Timer** routines, but these capabilities are not needed by most drivers.

First, the **KTIMER** structure used as a timer object by the **Ke*Xxx*Timer** routines is driver-allocated. The driver can preallocate this object to ensure that the object is available even in circumstances in which resources are constrained and memory allocations can fail. In contrast, a call to **ExAllocateTimer** to allocate a timer object might fail in a resource-constrained environment. However, few drivers need to be designed to operate in environments in which memory allocations fail, and most drivers benefit from the convenience of an **ExAllocateTimer** routine that both allocates and initializes a timer object.

Second, there is no **Ex*Xxx*Timer** equivalent of the **KeReadStateTimer** routine, which indicates whether a timer object is in the signaled state. However, this routine is rarely used. If necessary, a driver that uses the **Ex*Xxx*Timer** routines can check whether a timer object is in the signaled state by reading a Boolean value that is set by the *ExTimerCallback* callback routine that the driver supplies to the **ExAllocateTimer** routine.

High-Resolution Timers

6/25/2019 • 4 minutes to read • Edit Online

Starting with Windows 8.1, drivers can use the **Ex*Xxx*Timer** routines to manage high-resolution timers. The accuracy of a high-resolution timer is limited only by the maximum supported resolution of the system clock. In contrast, timers that are limited to the default system clock resolution are significantly less accurate.

However, high-resolution timers require system clock interrupts to—at least, temporarily—occur at a higher rate, which tends to increase power consumption. Thus, drivers should use high-resolution timers only when timer accuracy is essential, and use default-resolution timers in all other cases.

To create a high-resolution timer, a WDM driver calls the **ExAllocateTimer** routine and sets the EX_TIMER_HIGH_RESOLUTION flag in the *Attributes* parameter. When the driver calls the **ExSetTimer** routine to set the high-resolution timer, the operating system increases the resolution of the system clock, as necessary, so that the times at which the timer expires more precisely correspond to the nominal expiration times specified in the *DueTime* and *Period* parameters.

A Kernel-Mode Driver Framework (KMDF) driver can call the **WdfTimerCreate** method to create a high-resolution timer. In this call, the driver passes a pointer to a **WDF_TIMER_CONFIG** structure as a parameter. To create a high-resolution timer, the driver sets the **UseHighResolutionTimer** member of this structure to **TRUE**. This member is a part of the structure starting with Windows 8.1 and KMDF version 1.13.

Controlling timer accuracy

For example, for Windows running on an x86 processor, the default interval between system clock ticks is typically about 15 milliseconds, and the minimum interval between system clock ticks is about 1 millisecond. Thus, the expiration time of a default-resolution timer (which **ExAllocateTimer** creates if the EX_TIMER_HIGH_RESOLUTION flag is not set) can be controlled only to within about 15 milliseconds, but the expiration time of a high-resolution timer can be controlled to within a millisecond.

If a driver specifies a relative expiration time for a default-resolution timer, the timer can expire up to about 15 milliseconds earlier or later than the specified expiration time. If a driver specifies a relative expiration time for a high-resolution timer, the timer can expire as late as about a millisecond after the specified expiration time but it never expires early. For more information about the relationship between system clock resolution and timer accuracy, see Timer Accuracy.

If no high-resolution timers are set, the operating system typically runs the system clock at its default rate. However, if one or more high-resolution timers are set, the operating system might need to run the system clock at its maximum rate for at least a part of the time before these timers expire.

To avoid unnecessarily increasing power consumption, the operating system runs the system clock at its maximum rate only when necessary to satisfy the timing requirements of high-resolution timers. For example, if a high-resolution timer is periodic, and its period spans several default system clock ticks, the operating system might run the system clock at its maximum rate only in the part of the timer period that immediately precedes each expiration. For the rest of the timer period, the system clock runs at its default rate.

To prevent excessive power consumption, drivers should avoid setting the period of a long-running high-resolution timer to a value that is less than the default interval between system clock ticks. Otherwise, the operating system is forced to continuously run the system clock at its maximum rate.

Starting with Windows 8, a driver can call the **ExQueryTimerResolution** routine to get the range of timer resolutions that are supported by the system clock.

Comparison to ExSetTimerResolution

Starting with Windows 2000, a driver can call the **ExSetTimerResolution** routine to change the time interval between successive system clock interrupts. For example, a driver can call this routine to change the system clock from its default rate to its maximum rate to improve timer accuracy. However, using **ExSetTimerResolution** has several disadvantages compared to using high-resolution timers created by **ExAllocateTimer**.

First, after calling **ExSetTimerResolution** to temporarily increase the system clock rate, a driver must call **ExSetTimerResolution** a second time to restore the system clock to its default rate. Otherwise, the system clock timer continuously generates interrupts at the maximum rate, which might cause excessive power consumption.

Second, a driver that uses the **ExSetTimerResolution** routine cannot optimize its temporary use of higher system clock rates as effectively as the operating system does for high-resolution timers. Thus, the system clock spends more time running at the maximum rate than is strictly necessary.

Third, if multiple drivers concurrently use **ExSetTimerResolution** to improve timer accuracy, the system clock might run at its maximum rate for long periods. In contrast, the operating system globally coordinates the operation of multiple high-resolution timers so that the system clock runs at the maximum rate only when necessary to meet the timing requirements of these timers.

Finally, using **ExSetTimerResolution** is inherently less accurate than using a high-resolution timer. After a driver calls **ExSetTimerResolution** to increase the system clock to its maximum rate, which is typically about a tick per millisecond, the driver might call a routine such as **KeSetTimerEx** to set the timer. If, in this call, the driver specifies a relative expiration time, the timer can expire up to about a millisecond earlier than or later than the specified expiration time. However, if a relative expiration time is specified for a high-resolution timer, the timer can expire up to about a millisecond later than the specified expiration time but it never expires early.

No-Wake Timers

6/25/2019 • 3 minutes to read • Edit Online

Starting with Windows 8.1, drivers can use no-wake timers to avoid unnecessarily waking a processor from a low-power state. By keeping the processor in a low-power state, a no-wake timer reduces power consumption and extends the time that a tablet or other mobile computer can run on a battery charge.

A timer can expire only when the processor is in an active, running state. If a timer reaches its expiration time when the processor is in a low-power state, and the timer needs to expire immediately, the timer must wake the processor. However, when a no-wake timer reaches its expiration time and the processor is in a low-power state, this timer waits to expire until the processor wakes for some reason other than the timer. As an option, a driver can specify a maximum delay tolerance for a no-wake timer so that if the processor does not wake (for some other reason) within the maximum delay tolerance after the timer's expiration time, the timer wakes the processor.

A driver can use a no-wake timer to initiate noncritical operations that need to be performed only when the processor is in an active state. For example, a driver might use a no-wake timer to periodically flush accumulated status information from a memory buffer to a file. This status information describes processing work that the driver performs only when the processor is active. When the processor is in a low-power state, no status information is generated, and there is no need to wake the processor.

To create a no-wake timer, a WDM driver calls the **ExAllocateTimer** routine. In this call, the driver sets the EX_TIMER_NO_WAKE flag bit in the *Attributes* parameter.

To set a no-wake timer to expire at some due time, the driver calls the **ExSetTimer** routine. In this call, the driver can specify how long the no-wake timer should wait after it reaches its expiration time before the timer wakes the processor. The driver writes this tolerable-delay time to the **NoWakeTolerance** member in the **EXT_SET_PARAMETERS** structure that the driver passes as an input parameter to the **ExSetTimer** routine. If the driver sets the **NoWakeTolerance** member to the special value EX_TIMER_UNLIMITED_TOLERANCE, the timer never wakes the processor and, thus, cannot expire until the processor wakes for some other reason.

A Kernel-Mode Driver Framework (KMDF) driver or User-Mode Driver Framework (UMDF) driver can call the **WdfTimerCreate** method to create a no-wake timer. In this call, the driver passes a pointer to a **WDF_TIMER_CONFIG** structure as a parameter. To create a no-wake timer that never wakes the processor, the driver sets the **TolerableDelay** member of this structure to the **TolerableDelayUnlimited** constant. This constant is supported starting with Windows 8.1 and KMDF version 1.13 or UMDF 2.0.

Comparison to coalescable timers

The **KeSetCoalescableTimer** routine was introduced in Windows 7. This routine enables a driver to specify how much tolerance to allow in the expiration time of a timer. Frequently, the operating system can use this information to coalesce two or more timer interrupts into a single interrupt. If the expiration times of multiple timers are close enough to each other that their tolerance windows overlap, a single timer interrupt in the region of overlap can satisfy the timing requirements of all of these timers.

The chief benefit of timer coalescing is that it extends the time that the processor can stay in a low-power state between timer expirations. Thus, drivers use timer coalescing and no-wake timers for similar purposes.

However, coalesceable timers behave differently from no-wake timers. In particular, the tolerable delay specified for a no-wake timer applies only when the processor is in a low-power state, whereas the tolerance specified for the expiration of a coalescable timer applies regardless of whether the processor is in a low-power state. For a coalescable timer, a driver can increase the amount of tolerance in the expiration time to reduce the likelihood that the timer wakes the processor, but increasing the tolerance has the side effect of decreasing the accuracy of the

timer when the processor is active. In contrast, the tolerable delay specified for a no-wake timer does not affect the accuracy of the timer when the processor is active. For many drivers, no-wake timers might be a better way to reduce power consumption.

Deleting a System-Allocated Timer Object

6/25/2019 • 3 minutes to read • Edit Online

Starting with Windows 8.1, the **ExDeleteTimer** routine deletes a timer object that was created by the **ExAllocateTimer** routine. This timer object is a system-allocated **EX_TIMER** structure whose members are opaque to drivers. Before a timer object is deleted, **ExDeleteTimer** disables further timer operations on the object, and cancels or completes any pending operation on the object that might be in progress.

After a driver calls **ExDeleteTimer**, this routine takes several steps to ensure that it can safely delete the timer object. First, **ExDeleteTimer** marks the timer object as disabled to prevent the driver from starting a new timer operation that uses the object. After the timer object is disabled, a call to the **ExSetTimer** or **ExCancelTimer** routine immediately returns **FALSE** and performs no operation. Also, a second call to **ExDeleteTimer** returns **FALSE** and performs no operation.

Next, **ExDeleteTimer** checks whether a timer is still pending from a previous call to **ExDeleteTimer**. Disabling a timer object does not cancel a timer that was set before the object was disabled. In either of the following two cases, a timer that was previously set might expire after the timer object is disabled:

- The timer is periodic.
- The timer is one-shot (or nonperiodic) and has not yet expired.

A periodic timer can never expire more than once after the timer object is disabled.

If your driver implements an *ExTimerCallback* callback routine, the *Timer* parameter to this routine is guaranteed to always be a valid pointer to the timer object (an **EX_TIMER** structure), even if the timer expires after the timer object is disabled.

If no timer is pending, **ExDeleteTimer** deletes the timer object and returns without waiting.

If a timer is pending when **ExDeleteTimer** is called, the *Cancel* and *Wait* parameter values that your driver supplies to this routine control the routine's behavior. The *Cancel* parameter tells **ExDeleteTimer** whether to try to cancel a pending timer. The *Wait* parameter tells **ExDeleteTimer** whether to wait to return until the timer object is deleted.

If *Cancel* is **FALSE** (in which case, *Wait* must be **FALSE**) and a timer is pending, **ExDeleteTimer** lets the timer expire before the timer object is deleted. In this case, **ExDeleteTimer** marks the timer object to indicate that it is to be deleted after the pending timer expires (and any last callback to the *ExTimerCallback* routine finishes). Then **ExDeleteTimer** returns without waiting either for the timer to finish expiring or for the object to be deleted.

If *Cancel* is **TRUE**, **ExDeleteTimer** tries to cancel a pending timer before it expires. **ExDeleteTimer** returns **TRUE** if it successfully cancels the timer. **ExDeleteTimer** returns **FALSE** if it cannot cancel the timer, which is the case for a one-shot timer that has already expired or is in the process of expiring. **ExDeleteTimer** also returns **FALSE** if the (one-shot or periodic) timer was canceled before the **ExDeleteTimer** call or if the timer was never set.

If *Cancel* is **TRUE** and *Wait* is **FALSE**, **ExDeleteTimer** never blocks the calling thread. If the timer object cannot be immediately deleted, **ExDeleteTimer** marks the timer object to indicate that it is to be deleted after the pending timer finishes expiring, and returns immediately without waiting either for the timer to expire or for the object to be deleted.

If *Cancel* and *Wait* are both **TRUE**, **ExDeleteTimer** blocks the calling thread if the timer object cannot be immediately deleted. **ExDeleteTimer** waits, if necessary, for the timer to finish expiring and for any callback to a driver-implemented *ExTimerCallback* routine to finish. Next, **ExDeleteTimer** deletes the timer object and calls the

ExTimerDeleteCallback routine, if the driver implements this routine. Finally, **ExDeleteTimer** returns.

A driver can call **ExDeleteTimer** from the driver's *ExTimerCallback* routine, which runs at IRQL = DISPATCH_LEVEL, but the driver must set the *Wait* parameter in this call to **FALSE**.

As an option, a driver can implement an *ExTimerDeleteCallback* callback routine that runs after a timer object is deleted. Typically, an *ExTimerDeleteCallback* routine frees any system resources that the driver allocated to use with the timer object.

ExDeleteTimer schedules a driver-implemented *ExTimerDeleteCallback* routine to run after the timer object is deleted, at which time the pointer to this object is no longer valid. If the *Wait* parameter is **TRUE** in the **ExDeleteTimer** call, the callback to the *ExTimerDeleteCallback* routine finishes before **ExDeleteTimer** returns. If *Wait* is **FALSE**, the *ExTimerDeleteCallback* routine might run before or after **ExDeleteTimer** returns.

For more information, see Ex*Xxx*Timer Routines and EX_TIMER Objects.

Event Objects

12/5/2018 • 2 minutes to read • Edit Online

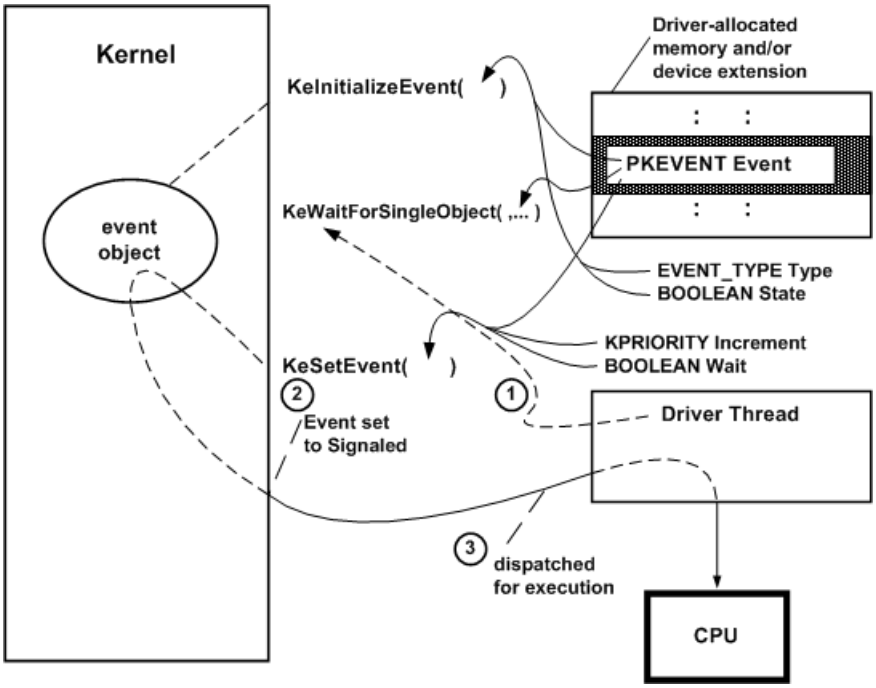
A driver can use an event object to wait while the next-lower driver processes an IRP set up by the waiting driver. Drivers that have driver-created threads or driver dispatch routines that wait for the completion of a synchronous I/O request also can use an event object to synchronize operations between their threads and/or other driver routines.

This section contains the following topics:

Defining and Using an Event Object

6/25/2019 • 3 minutes to read • Edit Online

Any driver that uses an event object must call **KeInitializeEvent**, **IoCreateNotificationEvent**, or **IoCreateSynchronizationEvent** before it waits on, sets, clears, or resets the event. The following figure illustrates how a driver with a thread can use an event object for synchronization.



As the previous figure shows, such a driver must provide the storage for the event object, which must be resident. The driver can use the device extension of a driver-created device object, the controller extension if it uses a controller object, or nonpaged pool allocated by the driver.

When the driver calls **KeInitializeEvent**, it must pass a pointer to the driver's resident storage for the event object. In addition, the caller must specify the initial state (signaled or not signaled) for the event object. The caller also must specify the event type, which can be either of the following:

- **SynchronizationEvent**

  When a *synchronization event* is set to the Signaled state, a single thread that is waiting for the event to be reset to Not-Signaled becomes eligible for execution and the event's state is automatically reset to Not-Signaled.

  This type of event is sometimes called an *autoclearing event*, because its Signaled state is automatically reset each time a wait is satisfied.

- **NotificationEvent**

  When a *notification event* is set to the Signaled state, all threads that were waiting for the event to be reset to Not-Signaled become eligible for execution and the event remains in the Signaled state until an explicit reset to Not-Signaled occurs: that is, there is a call to **KeClearEvent** or **KeResetEvent** with the given *Event* pointer.

Few device or intermediate drivers have a single driver-dedicated thread, let alone a set of threads that might synchronize their operations by waiting for an event that protects a shared resource.

Most drivers that use event objects to wait for the completion of an I/O operation set the input *Type* to

NotificationEvent when they call **KeInitializeEvent**. An event object set up for IRPs that a driver creates with **IoBuildSynchronousFsdRequest** or **IoBuildDeviceIoControlRequest** is almost always initialized as a **NotificationEvent** because the caller will wait for the event for notification that its request has been satisfied by one or more lower-level drivers.

After the driver has initialized itself, its driver-dedicated thread, if any, and other routines can synchronize their operations on the event. For example, a driver with a thread that manages the queuing of IRPs, such as the system floppy controller driver, might synchronize IRP processing on an event, as shown in the previous figure:

1. The thread, which has dequeued an IRP for processing on the device, calls **KeWaitForSingleObject** with a pointer to the driver-supplied storage for the initialized event object.

2. Other driver routines carry out device the I/O operations necessary to satisfy the IRP and, when these operations are complete, the driver's *DpcForIsr* routine calls **KeSetEvent** with a pointer to the event object, a driver-determined priority boost for the thread (*Increment*, as shown in the previous figure), and a Boolean *Wait* set to **FALSE**. Calling **KeSetEvent** sets the event object to the Signaled state, thereby changing the waiting thread's state to ready.

3. The kernel dispatches the thread for execution as soon as a processor is available: that is, no other thread with a higher priority is currently in the ready state and there are no kernel-mode routines to be run at a higher IRQL.

   The thread now can complete the IRP if the *DpcForIsr* has not called **IoCompleteRequest** with the IRP already, and can dequeue another IRP to be processed on the device.

Calling **KeSetEvent** with the *Wait* parameter set to **TRUE** indicates the caller's intention to immediately call a **KeWaitForSingleObject** or **KeWaitForMultipleObjects** support routine on return from **KeSetEvent**.

Consider the following guidelines for setting the*Wait*parameter toKeSetEvent:

A pageable thread or pageable driver routine that runs at IRQL < DISPATCH_LEVEL should never call **KeSetEvent** with the *Wait* parameter set to **TRUE**. Such a call causes a fatal page fault if the caller happens to be paged out between the calls to **KeSetEvent** and **KeWaitForSingleObject** or **KeWaitForMultipleObjects**.

Any standard driver routine that runs at IRQL = DISPATCH_LEVEL cannot wait for a nonzero interval on any dispatcher objects without bringing down the system. However, such a routine can call **KeSetEvent** while running at an IRQL less than or equal to DISPATCH_LEVEL.

For a summary of the IRQLs at which standard driver routines run, see Managing Hardware Priorities.

KeResetEvent returns the previous state of a given *Event*: whether it was set to Signaled or not when the call to **KeResetEvent** occurred. **KeClearEvent** simply sets the state of the given *Event* to Not-Signaled.

Consider the following guideline for when to call the preceding support routines:

For better performance, every driver should call **KeClearEvent** unless the caller needs the information returned by **KeResetEvent** to determine what to do next.

Standard Event Objects

6/25/2019 • 2 minutes to read • Edit Online

The system provides several standard event objects. Drivers can use these event objects to be notified by the system whenever certain conditions occur. The following list contains the standard event objects:

\KernelObjects\HighMemoryCondition
This event is set whenever the amount of free physical memory exceeds a system-defined amount. Drivers can wait for this event to be set as a signal to aggressively allocate memory.

\KernelObjects\LowMemoryCondition
This event is set whenever the amount of free physical memory falls below a system-defined amount. Drivers that have allocated large amounts of memory can wait for this event to be set as a signal to free unused memory.

For Microsoft Windows Server 2003 and later versions of Windows, drivers can also use the following additional standard event objects:

\KernelObjects\HighPagedPoolCondition
This event is set whenever the amount of free paged pool exceeds a system-defined amount. Drivers can wait for this event to be set as a signal to aggressively allocate memory from paged pool.

\KernelObjects\LowPagedPoolCondition
This event is set whenever the amount of free paged pool falls below a system-defined amount. Drivers that have allocated large amounts of memory can wait for this event to be set as a signal to free unused memory from paged pool.

\KernelObjects\HighNonPagedPoolCondition
This event is set whenever the amount of free nonpaged pool exceeds a system-defined amount. Drivers can wait for this event to be set as a signal to aggressively allocate memory from nonpaged pool.

\KernelObjects\LowNonPagedPoolCondition
This event is set whenever the amount of free nonpaged pool falls below a system-defined amount. Drivers that have allocated large amounts of memory can wait for this event to be set as a signal to free unused memory from nonpaged pool.

For Windows Vista and later versions of Windows, drivers can also use the following additional standard event objects:

\KernelObjects\LowCommitCondition
This event is set when the operating system's *commit charge* is low, relative to the *current commit limit*. In other words, memory usage is low and a lot of space is available in physical memory or paging files.

\KernelObjects\HighCommitCondition
This event is set when the operating system's commit charge is high, relative to the current commit limit. In other words, memory usage is high and very little space is available in physical memory or paging files, but the operating system might be able to increase the size of its paging files.

\KernelObjects\MaximumCommitCondition
This event is set when the operating system's commit charge is near the *maximum commit limit*. In other words, memory usage is very high, very little space is available in physical memory or paging files, and the operating system cannot increase the size of its paging files. (A system administrator can always increase the size or number of paging files, without restarting the computer, if sufficient storage resources exist.)

Each of these events are notification events. They remain set as long as the triggering condition remains true.

To open a handle to any of these events, use the **IoCreateNotificationEvent** routine. A driver that waits for any of these events should create a dedicated thread to do the waiting. The thread can wait for one or more of these events by calling either **KeWaitForSingleObject** or **KeWaitForMultipleObjects**.

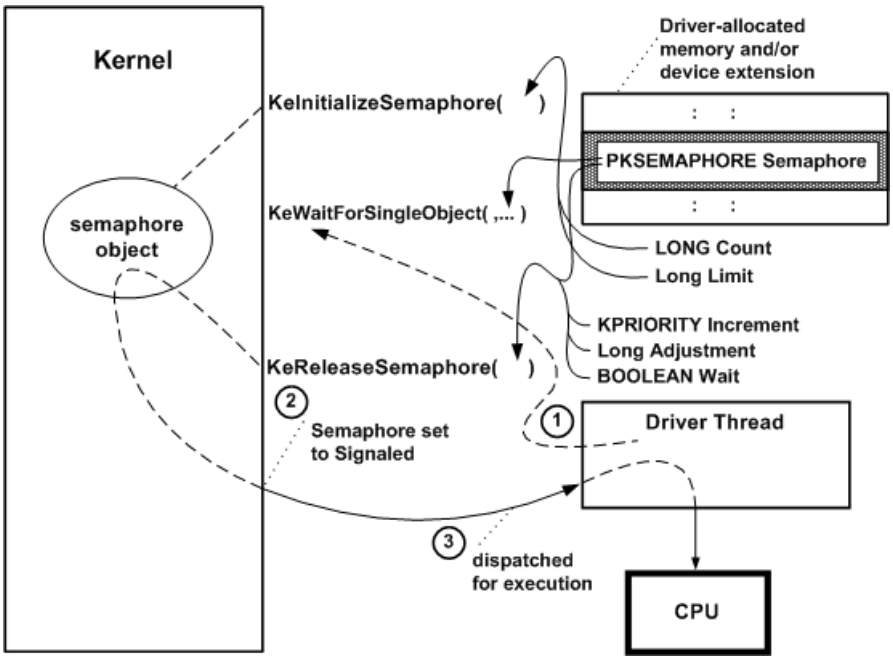
Semaphore Objects

6/25/2019 • 4 minutes to read • Edit Online

Any driver can use a semaphore object to synchronize operations between its driver-created threads and other driver routines. For example, a driver-dedicated thread might put itself into a wait state when there are no outstanding I/O requests for the driver, and the driver's dispatch routines might set the semaphore to the Signaled state just after they queue an IRP.

The dispatch routines of highest-level drivers, which are run in the context of the thread requesting an I/O operation, might use a semaphore to protect a resource shared among the dispatch routines. Lower-level driver dispatch routines for synchronous I/O operations also might use a semaphore to protect a resource shared among that subset of dispatch routines or with a driver-created thread.

Any driver that uses a semaphore object must call **KeInitializeSemaphore** before it waits on or releases the semaphore. The following figure illustrates how a driver with a thread can use a semaphore object.



As the previous figure shows, such a driver must provide the storage for the semaphore object, which should be resident. The driver can use the device extension of a driver-created device object, the controller extension if it uses a controller object, or nonpaged pool allocated by the driver.

When the driver's *AddDevice* routine calls **KeInitializeSemaphore**, it must pass a pointer to the driver's resident storage for the semaphore object. In addition, the caller must specify a *Count* for the semaphore object, as shown in the previous figure, that determines its initial state (nonzero for Signaled).

The caller also must specify a *Limit* for the semaphore, which can be either of the following:

- **Limit = 1**

  When this semaphore is set to the Signaled state, a single thread waiting for the semaphore to be reset to the Not-Signaled state becomes eligible for execution and can access whatever resource is protected by the semaphore.

  This type of semaphore is also called a *binary semaphore* because a thread either has or does not have exclusive access to the semaphore-protected resource.

- **Limit > 1**

  When this semaphore is set to the Signaled state, some number of threads waiting for the semaphore object to be set to the Not-Signaled state become eligible for execution and can access whatever resource is protected by the semaphore.

  This type of semaphore is called a *counting semaphore* because the routine that sets the semaphore to the Signaled state also specifies how many waiting threads can have their states changed from waiting to ready. The number of such waiting threads can be the *Limit* set when the semaphore was initialized or some number less than this preset *Limit*.

Few device or intermediate drivers have a single driver-created thread; even fewer have a set of threads that might wait for a semaphore to be acquired or released. Few system-supplied drivers use semaphore objects, and, of those that do, even fewer use a binary semaphore. Although a binary semaphore might seem to be similar in functionality to a mutex object, a binary semaphore does not provide the built-in protection against deadlocks that a mutex object has for system threads running in SMP machines.

After a driver with an initialized semaphore is loaded, it can synchronize operations on the semaphore that protects a shared resource. For example, a driver with a device-dedicated thread that manages the queuing of IRPs, such as the system floppy controller driver, might synchronize IRP queuing on a semaphore, as shown in the previous figure:

1. The thread calls **KeWaitForSingleObject** with a pointer to the driver-supplied storage for the initialized semaphore object to put itself into a wait state.

2. IRPs begin to come in that require device I/O operations. The driver's dispatch routines insert each such IRP into an interlocked queue under spin-lock control and call **KeReleaseSemaphore** with a pointer to the semaphore object, a driver-determined priority boost for the thread (*Increment*, as shown in the previous figure), an *Adjustment* of 1 that is added to the semaphore's Count as each IRP is queued, and a Boolean *Wait* set to **FALSE**. A nonzero semaphore Count sets the semaphore object to the Signaled state, thereby changing the waiting thread's state to ready.

3. The kernel dispatches the thread for execution as soon as a processor is available: that is, no other thread with a higher priority is currently in the ready state and there are no kernel-mode routines to be run at a higher IRQL.

   The thread removes an IRP from the interlocked queue under spin-lock control, passes it on to other driver routines for further processing, and calls **KeWaitForSingleObject** again. If the semaphore is still set to the Signaled state (that is, its Count remains nonzero, indicating that more IRPs are in the driver's interlocked queue), the kernel again changes the thread's state from waiting to ready.

   By using a counting semaphore in this manner, such a driver thread "knows" there is an IRP to be removed from the interlocked queue whenever that thread is run.

Calling **KeReleaseSemaphore** with the *Wait* parameter set to **TRUE** indicates the caller's intention to immediately call a **KeWait*Xxx*Object**(s) support routine on return from **KeReleaseSemaphore**.

Consider the following guidelines for setting the *Wait* parameter to KeReleaseSemaphore:

A pageable thread or pageable driver routine that runs at IRQL PASSIVE_LEVEL should never call **KeReleaseSemaphore** with the *Wait* parameter set to **TRUE**. Such a call causes a fatal page fault if the caller happens to be paged out between the calls to **KeReleaseSemaphore** and **KeWait*Xxx*Object**(s).

Any standard driver routine that runs at an IRQL greater than PASSIVE_LEVEL cannot wait for a nonzero interval on any dispatcher objects without bringing down the system; see Kernel Dispatcher Objects for details. However, such a routine can call **KeReleaseSemaphore** while running at an IRQL less than or equal to DISPATCH_LEVEL.

For a summary of the IRQLs at which standard driver routines run, see Managing Hardware Priorities. For IRQL

requirements of a specific support routine, see the routine's reference page.

Introduction to Mutex Objects

6/25/2019 • 4 minutes to read • Edit Online

As its name suggests, a mutex object is a synchronization mechanism designed to ensure mutually exclusive access to a single resource that is shared among a set of kernel-mode threads. Only highest-level drivers, such as file system drivers (FSDs) that use executive worker threads, are likely to use a mutex object.

Possibly, a highest-level driver with driver-created threads or worker-thread callback routines might use a mutex object. However, any driver with pageable threads or worker-thread callback routines must manage the acquisitions of, waits on, and releases of its mutex objects very carefully.

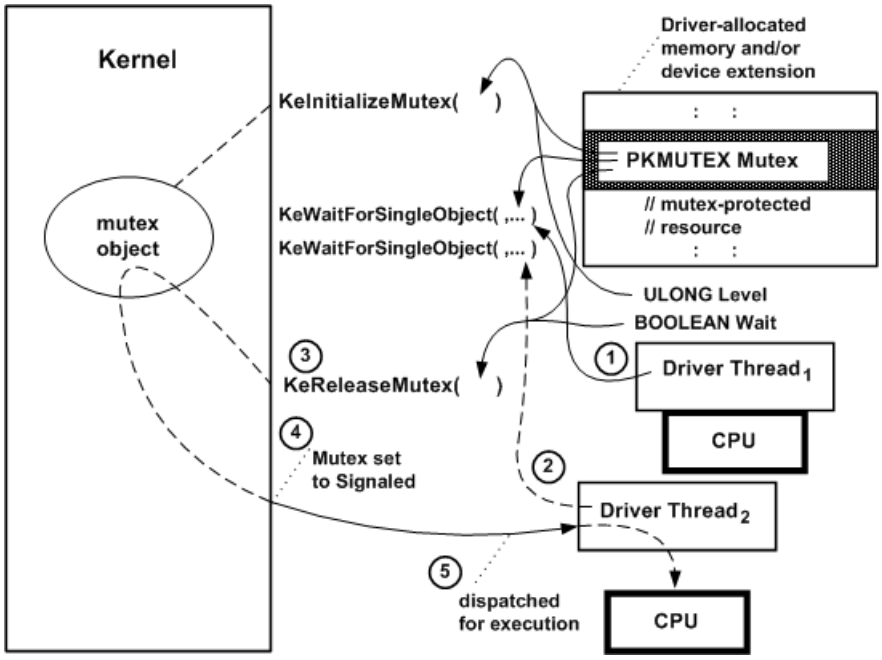
Mutex objects have built-in features that provide system (kernel-mode only) threads mutually exclusive, deadlock-free access to shared resources in SMP machines. The kernel assigns ownership of a mutex to a single thread at a time.

Acquiring ownership of a mutex prevents the delivery of normal kernel-mode asynchronous procedure calls (APCs). The thread will not be preempted by an APC unless the kernel issues an APC_LEVEL software interrupt to run a special kernel APC, such as the I/O manager's IRP completion routine that returns results to the original requester of an I/O operation

A thread can acquire ownership of a mutex object that it already owns (recursive ownership), but a recursively acquired mutex object is not set to the Signaled state until the thread releases its ownership completely. Such a thread must explicitly release the mutex as many times as it acquired ownership before another thread can acquire the mutex.

The kernel never permits a thread that owns a mutex to cause a transition to user mode without first releasing the mutex and setting it to the Signaled state. If any FSD-created or driver-created thread that owns a mutex attempts to return control to the I/O manager before releasing ownership of the mutex, the kernel brings down the system.

Any driver that uses a mutex object must call **KeInitializeMutex** once before it waits on or releases its mutex object. The following figure illustrates how two system threads might use a mutex object.



As the previous figure shows, a driver that uses a mutex object must provide the storage for the mutex object, which must be resident. The driver can use the device extension of a driver-created device object, the controller

extension if it uses a controller object, or nonpaged pool that is allocated by the driver.

When a driver calls **KeInitializeMutex** (typically from its *AddDevice* routine), it must pass a pointer to the driver's storage for the mutex object, which the kernel initializes to the Signaled state.

After such a highest-level driver has initialized, it can manage mutually exclusive access to a shared resource as shown in the previous figure. For example, a driver's dispatch routines for inherently synchronous operations and threads might use a mutex to protect a driver-created queue for IRPs.

Because **KeInitializeMutex**always sets the initial state of a mutex object to Signaled (as the previous figure shows):

1. A dispatch routine's initial call to **KeWaitForSingleObject** with the *Mutex* pointer puts the current thread immediately into the ready state, gives the thread ownership of the mutex, and resets the mutex state to Not-Signaled. As soon as the dispatch routine resumes running, it can safely insert an IRP into the mutex-protected queue.

2. When a second thread (another dispatch routine, driver-supplied worker-thread callback routine, or driver-created thread) calls **KeWaitForSingleObject** with the *Mutex* pointer, the second thread is put into the wait state.

3. When the dispatch routine finishes queuing the IRP as described in step 1, it calls **KeReleaseMutex** with the *Mutex* pointer and a Boolean *Wait* value, which indicates whether it intends to call **KeWaitForSingleObject** (or **KeWaitForMutexObject**) with the *Mutex* as soon as **KeReleaseMutex** returns control.

4. Assuming the dispatch routine released its ownership of the mutex in step 3 (*Wait* set to **FALSE**), the mutex is set to the Signaled state by **KeReleaseMutex**. The mutex currently has no owner, so the kernel determines whether another thread is waiting for that mutex. If so, the kernel makes the second thread (see step 2) the mutex owner, possibly boosts the thread's priority to the lowest real-time priority value, and changes its state to ready.

5. The kernel dispatches the second thread for execution as soon as a processor is available: that is, when no other thread with a higher priority is currently in the ready state and there are no kernel-mode routines to be run at a higher IRQL. The second thread (a dispatch routine queuing an IRP or the driver's worker-thread callback routine or driver-created thread dequeuing an IRP) can now safely access the mutex-protected queue of IRPs until it calls **KeReleaseMutex**.

If a thread acquires ownership of a mutex object recursively, that thread must explicitly call **KeReleaseMutex** as many times as it waited on the mutex in order to set the mutex object to the Signaled state. For example, if a thread calls **KeWaitForSingleObject** and then **KeWaitForMutexObject** with the same *Mutex* pointer, it must call **KeReleaseMutex** twice when it acquires the mutex in order to set that mutex object to the Signaled state.

Calling **KeReleaseMutex** with the *Wait* parameter set to **TRUE** indicates the caller's intention to immediately call a **KeWait*Xxx*** support routine on return from **KeReleaseMutex**.

Consider the following guidelines for setting the Wait parameter to KeReleaseMutex:

A pageable thread or pageable driver routine that runs at IRQL PASSIVE_LEVEL should never call **KeReleaseMutex** with the *Wait* parameter set to **TRUE**. Such a call causes a fatal page fault if the caller happens to be paged out between the calls to **KeReleaseMutex** and **KeWait*Xxx*Object**(s).

Any standard driver routine that runs at an IRQL greater than PASSIVE_LEVEL cannot wait for a nonzero interval on any dispatcher objects without bringing down the system. However, such a routine can call **KeReleaseMutex** if it owns the mutex while running at an IRQL less than or equal to DISPATCH_LEVEL.

For a summary of the IRQLs at which standard driver routines run, see Managing Hardware Priorities.

Alternatives to Mutex Objects

6/25/2019 • 2 minutes to read • Edit Online

Fast mutexes and guarded mutexes can be used as a replacement for mutex objects. A fast mutex or guarded mutex can be acquired and released faster than a mutex object, but they have the following restrictions:

- Drivers cannot use the **KeWaitForSingleObject** or **KeWaitForMultipleObjects** routines to wait for a fast or guarded mutex. Thus, a driver cannot wait for a fast or guarded mutex and a dispatcher object simultaneously.
- Drivers cannot acquire a fast or guarded mutex recursively. If a driver tries to acquire a fast or guarded mutex that it has already acquired, the driver will deadlock. A mutex object, however, can be acquired recursively.

For more information about fast and guarded mutexes, see Fast Mutexes and Guarded Mutexes.

Introduction to Thread Objects

1/11/2019 • 2 minutes to read • Edit Online

A user-mode thread object represents a path of execution within the current process. Every user-mode thread object is implemented through the use of an embedded kernel-mode thread object.

A kernel-mode thread object is an instance of a kernel-defined dispatcher object type. The thread that it represents is the basic schedulable entity in the operating system.

A thread object:

- Is dispatched for execution by the kernel.
- Has the following properties at any given moment:
  - *dispatch state*
  - *priority*
  - *context*
  - Execution mode (kernel or user)
  - *affinity*
- Is "owned by" a process object but can attach itself to another process's address space.

Usually, most drivers execute in the context of the currently running thread, that is, in an arbitrary thread context. While a file system driver can create an independent process for its own device-dedicated threads, file systems usually avoid setting up a driver-created process and threads in order to conserve system memory and to avoid the overhead of context switches.

FSs (and other drivers) can set up device-dedicated (system-process) threads and/or FSs can use system worker threads if they need a driver-specific thread context in which to execute. Drivers use the kernel-mode **Ps*Xxx*** routines to create processes and/or device-dedicated threads. FSs call **Ex*Xxx*** routines to use system worker threads.

Thread Priorities

12/21/2018 • 2 minutes to read • Edit Online

Some drivers create their own driver- or device-dedicated system threads and set their thread's base priority to the lowest real-time priority value. Other highest-level drivers, particularly file system drivers, use system worker threads with a base priority that is usually set to the highest variable priority value. The kernel schedules a thread with the lowest real-time priority to run ahead of every thread with a variable priority, which includes almost every user-mode thread in the system.

Most standard driver routines are run in an arbitrary thread context, ahead of all threads that are currently in the ready state.

Threads, whatever their respective run-time priorities, are run at IRQL = PASSIVE_LEVEL. Many standard driver routines are run at an IRQL > PASSIVE_LEVEL, such as DISPATCH_LEVEL or DIRQL.

For more information about thread priorities, see the Scheduling, Thread Context, and IRQL white paper.

Device-Dedicated Threads

6/25/2019 • 2 minutes to read • Edit Online

The driver of a slow device or a device that is seldom used (like the floppy controller) can solve many "waiting" problems by creating a device-dedicated system thread. Additionally, most file system drivers use system worker threads and supply worker-thread callback routines.

If a device driver has its own thread context or is running in a system-thread context, the device-dedicated thread or highest-level driver's worker-thread callback routine can synchronize operations on a dispatcher object, such as an event object or semaphore object, in a shared communication region of the driver's device extension. For example, a device-dedicated thread can wait for a shared dispatcher object, while the thread's device is not in use, by calling **KeWaitForSingleObject** for a semaphore. Until the device driver is called to carry out an I/O operation (at which point it sets the semaphore to the Signaled state), its waiting thread uses no CPU time.

A driver can call **PsCreateSystemThread** to create a driver- or device-dedicated thread, and then call **KeSetBasePriorityThread** to set the thread's base priority. The driver should specify a priority value that avoids *run-time priority inversions* in SMP machines. That is, setting the base priority of a driver-created thread too high can create delays in the execution of lower priority threads that submit I/O requests for that driver.

Because thread objects are themselves a type of dispatcher object, a thread can wait for another thread to complete. To obtain the thread object pointer associated with a thread, a driver can call **ObReferenceObjectByHandle**, passing in the thread handle received from **PsCreateSystemThread**.

A thread can call **KeDelayExecutionThread** to wait for an interval that could be a full time slice or longer. The granularity of a **KeDelayExecutionThread** interval is around 10 milliseconds. Because **KeDelayExecutionThread** is a timer-driven routine, the granularity of its interval is slightly faster or slower than 10 milliseconds, depending on the platform.

System Worker Threads

6/25/2019 • 3 minutes to read • Edit Online

A driver that requires delayed processing can use a *work item*, which contains a pointer to a driver callback routine that performs the actual processing. The driver queues the work item, and a *system worker thread* removes the work item from the queue and runs the driver's callback routine. The system maintains a pool of these system worker threads, which are system threads that each process one work item at a time.

The driver associates a *WorkItem* callback routine with the work item. When the system worker thread processes the work item, it calls the associated *WorkItem* routine. In Windows Vista and later versions of Windows, a driver can instead associate a *WorkItemEx* routine with a work item. *WorkItemEx* takes parameters that are different from the parameters that *WorkItem* takes.

WorkItem and *WorkItemEx* routines run in a system thread context. If a driver dispatch routine can run in a user-mode thread context, that routine can call a *WorkItem* or *WorkItemEx* routine to perform any operations that require a system thread context.

To use a work item, a driver performs the following steps:

1. Allocate and initialize a new work item.

   The system uses an **IO_WORKITEM** structure to hold a work item. To allocate a new **IO_WORKITEM** structure and initialize it as a work item, the driver can call **IoAllocateWorkItem**. In Windows Vista and later versions of Windows, the driver can alternatively allocate its own **IO_WORKITEM** structure, and call **IoInitializeWorkItem** to initialize the structure as a work item. (The driver should call **IoSizeofWorkItem** to determine the number of bytes that are necessary to hold a work item.)

2. Associate a callback routine with the work item, and queue the work item so that it will be processed by a system worker thread.

   To associate a *WorkItem* routine with the work item and queue the work item, the driver should call **IoQueueWorkItem**. To instead associate a *WorkItemEx* routine with the work item and queue the work item, the driver should call **IoQueueWorkItemEx**.

3. After the work item is no longer required, free it.

   A work item that was allocated by **IoAllocateWorkItem** should be freed by **IoFreeWorkItem**. A work item that was initialized by **IoInitializeWorkItem** must be uninitialized by **IoUninitializeWorkItem** before it can be freed.

   The work item can only be uninitialized or freed when the work item is not currently queued. The system dequeues the work item before it calls the work item's callback routine, so **IoFreeWorkItem** and **IoUninitializeWorkItem** can be called from within the callback.

A DPC that needs to initiate a processing task that requires lengthy processing or that makes a blocking call should delegate the processing of that task to one or more work items. While a DPC runs, all threads are prevented from running. Additionally, a DPC, which runs at IRQL = DISPATCH_LEVEL, must not make blocking calls. However, the system worker thread that processes a work item runs at IRQL = PASSIVE_LEVEL. Thus, the work item can contain blocking calls. For example, a system worker thread can wait on a dispatcher object.

Because the pool of system worker threads is a limited resource, *WorkItem* and *WorkItemEx* routines can be used only for operations that take a short period of time. If one of these routines runs for too long (if it contains an indefinite loop, for example) or waits for too long, the system can deadlock. Therefore, if a driver requires long periods of delayed processing, it should instead call **PsCreateSystemThread** to create its own system thread.

Do not call **IoQueueWorkItem** or **IoQueueWorkItemEx** to queue a work item that is already in the queue. In checked builds, this error causes a bug check. In retail builds, the error is not detected but can cause corruption of system data structures. If your driver queues the same work item each time a particular driver routine runs, you can use the following technique to avoid queuing the work item a second time if it is already in the queue:

- The driver maintains a list of tasks for the worker routine.
- This task list is available in the context that is supplied to the worker routine. The worker routine and any driver routines that modify the task list synchronize their access to the list.
- Each time the worker routine runs, it performs all the tasks in the list, and removes each task from the list as the task is completed.
- When a new task arrives, the driver adds this task to the list. The driver queues the work item only if the task list was previously empty.

The system worker thread removes the work item from the queue before it calls the worker thread. Thus, a driver thread can safely queue the work item again as soon as the worker thread starts to run.

Waits and APCs

6/25/2019 • 2 minutes to read • Edit Online

Threads that wait for a dispatcher object on behalf of a user-mode caller must be prepared for that wait to be interrupted, either by a user APC or by thread termination. When a thread calls **KeWaitForSingleObject**, **KeWaitForMultipleObjects**, **KeWaitForMutexObject**, or **KeDelayExecutionThread**, the operating system can place the thread in a wait state. Typically, the thread remains in the wait state until the operating system can complete the operation that the caller requests. However, if the caller specifies *WaitMode* = UserMode, the operating system might interrupt the wait. In that case, the routine exits with an NTSTATUS value of STATUS_USER_APC.

Any driver that calls one of the preceding four routines with *WaitMode* = UserMode must be prepared to receive a return value of STATUS_USER_APC. The driver must complete its current operation with STATUS_USER_APC and return control to user mode.

The exact situations in which the operating system interrupts the wait depends on the value of the *Alertable* parameter of the routine. If *Alertable* = **TRUE**, the wait is an alertable wait. Otherwise, the wait is a non-alertable wait. The operating system interrupts alertable waits only to deliver a user APC. The operating system interrupts both kinds of waits to terminate the thread.

The following table explains the relationship between different parameter settings, waits, and user APC delivery.

PARAMETERS	WAIT INTERRUPTED?	USER APC DELIVERED?
Alertable = **TRUE** *WaitMode* = **UserMode**	Yes	Yes
Alertable = **TRUE** *WaitMode* = **KernelMode**	Yes	No
Alertable = **FALSE** *WaitMode* = **UserMode**	Yes, for thread termination. No, for user APCs.	No
Alertable = **FALSE** *WaitMode* = **KernelMode**	No	No

You can disable kernel APCs for a thread. If you do disable kernel APCs for a thread, both user APC delivery and thread termination for that thread are also disabled. For more information about how to disable APCs, see Disabling APCs.

Alerts, a seldom-used mechanism that are internal to the operating system, can also interrupt alertable wait states. An alert can interrupt a wait when *Alertable* = **TRUE**, regardless of the value of the *WaitMode* parameter. The waiting routine returns a value of STATUS_ALERTED.

Note that kernel APCs run preemptively, and do not cause **KeWaitFor*Xxx*** or **KeDelayExecutionThread** to return. The system interrupts and resumes the wait internally. Drivers are normally unaffected by this process, but it is possible for the driver to miss a dispatcher object signal for a transient condition, such as a call to **KePulseEvent**.

Callback Objects

12/5/2018 • 2 minutes to read • Edit Online

The kernel's callback mechanism provides a general way for drivers to request and provide notification when certain conditions are satisfied.

A driver can create a callback object, and other drivers can request notification for conditions associated with this driver-defined callback. In addition, the system defines two callback objects for driver use.

Every callback object has a name and a set of attributes, defined when the object is created. The system-defined callback objects are named **\Callback\SetSystemTime**, **\Callback\PowerState**, and **\Callback\ProcessorAdd**; driver-defined callbacks must not duplicate these names.

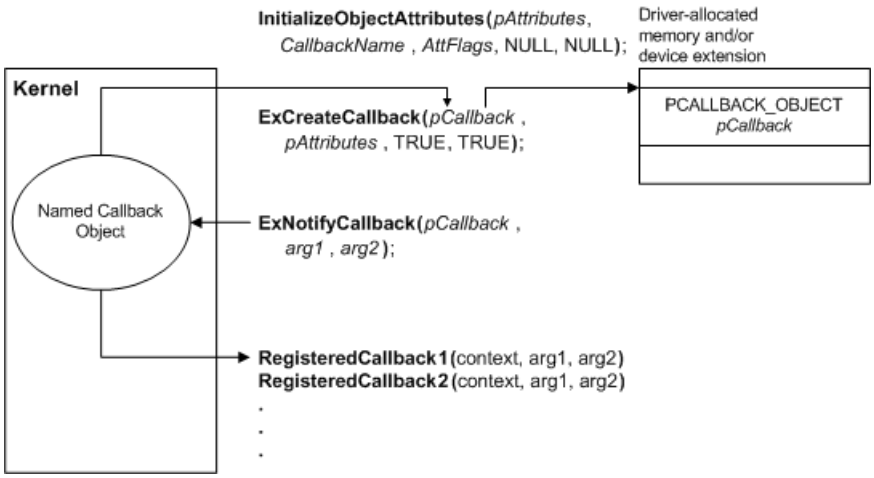
To request notification from a system- or driver-defined callback, a driver opens the callback object and registers a callback routine. When the conditions defined for the callback become true, its creator triggers notification. In turn, the system calls all the callback routines registered for the callback.

This section contains the following topics:

Defining a Callback Object

6/25/2019 • 2 minutes to read • Edit Online

A driver can create a callback object, through which other drivers can request notification of conditions defined by the creating driver. The following figure shows the steps involved in defining a callback object.



Before creating the object, the driver calls **InitializeObjectAttributes** to set its attributes. A callback object must have a name, which cannot match the name of a system-defined callback; it can have whatever other attributes its creator deems appropriate, typically OBJ_CASE_INSENSITIVE. Next the driver calls **ExCreateCallback**, passing a pointer to the initialized attributes and a location at which to receive a handle to the callback object. It also passes two Booleans, indicating whether the system should create the callback object if such a named object does not already exist, and whether the object should allow more than one registered callback routine.

The driver defines the conditions for which it will call the registered callback routines. The conditions take the form of two arguments, each pointing to a parameter defined by the driver that creates the callback. You should document these conditions, along with the name of the callback object and the IRQL at which it requests notification, for clients of the driver.

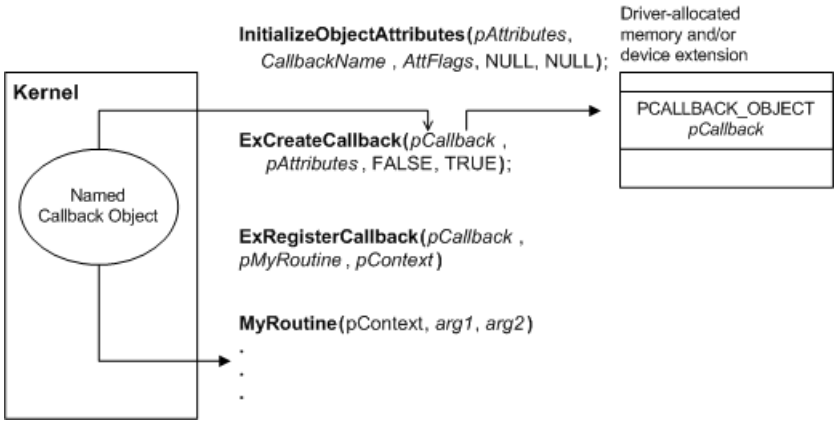
When the callback condition occurs, the driver calls **ExNotifyCallback**, passing its handle to the callback object and the two arguments. The system then calls all callback routines registered for the callback object, in the order in which they were registered, passing the two arguments and a pointer to the context supplied when the routine was registered. The driver must call **ExNotifyCallback** at IRQL <= DISPATCH_LEVEL; the system calls the callback routines at the same IRQL at which the driver made this call.

After all operations have been completed with the callback object, the driver that created the callback should call **ObDereferenceObject** to decrement its reference count and ensure that the object is deleted.

Using a Driver-Defined Callback Object

6/25/2019 • 2 minutes to read • Edit Online

To use a callback object defined by another driver, a driver opens the object, then registers a routine to be called when the callback is triggered, as shown in the following figure. The driver requesting notification must know the name of the callback object and must understand the semantics of the arguments passed to the callback routine.



Before it can open the object, the driver must call **InitializeObjectAttributes** to create an attribute block, specifying the name of the object. After it has a pointer to an attribute block, it calls **ExCreateCallback**, passing the attribute pointer, a location in which to receive a handle to the callback, and **FALSE** for the *Create* parameter, indicating that it requires an existing callback object.

The driver can then call **ExRegisterCallback** with the returned handle to register its callback routine.

The callback routine has the following prototype:

```
typedef VOID (*PCALLBACK_FUNCTION ) (
    IN PVOID CallbackContext,
    IN PVOID Argument1,
    IN PVOID Argument2
    );
```

The *CallbackContext* parameter is the context pointer to be passed to the callback routine each time it is called. Typically, this parameter is a pointer to a block of context data, which the caller should allocate from nonpaged pool if the routine can be called at DISPATCH_LEVEL. The two arguments are defined by the component that created the callback. Typically, the arguments provide information about the conditions that triggered the callback.

When the creator of the callback triggers notification, the system calls the registered routine, passing a pointer to the context and the two arguments. Values for the arguments are supplied by the component that created the callback. The callback routine is called at the same IRQL at which the creating driver triggered notification, which is always IRQL <= DISPATCH_LEVEL.

In its callback routine, a driver can perform whatever tasks it requires for the current conditions.

When the driver no longer requires notification, it should call **ExUnregisterCallback** to remove its routine from the list of registered callbacks and to remove its reference to the callback object.

Using a System-Defined Callback Object

6/25/2019 • 2 minutes to read • Edit Online

The system defines three callback objects for driver use:

\Callback\SetSystemTime

\Callback\PowerState

\Callback\ProcessorAdd

Drivers that use the system time (for example, file system drivers) might register for the **\Callback\SetSystemTime** callback object. This callback provides for notification when the system time changes.

The **\Callback\PowerState** callback object provides for notification when one of the following occurs:

- The system switches from AC to DC power or vice versa.
- The system power policy changes as the result of a user or application request.
- A transition to a system sleep or shutdown state is imminent. A driver can request notification so that it can lock code into memory in anticipation of shutdown. Callback routines will be notified before the power manager sends the system set-power IRP.

The **\Callback\ProcessorAdd** callback provides notification when a new processor is added to the system.

To use a system-defined callback, a driver initializes an attribute block (**InitializeObjectAttributes**) with the callback's name, then opens the callback object (**ExCreateCallback**), just as for a driver-defined callback. The driver should not request that the callback object be created.

With the handle returned by **ExCreateCallback**, the driver calls **ExRegisterCallback** to register a notification routine, passing a pointer to an arbitrary context and a pointer to its routine. A driver can register its callback routine any time. When the specified condition occurs, the system calls the registered callback routine at IRQL<=DISPATCH_LEVEL.

When the driver no longer requires notification, it should call **ExUnregisterCallback** to delete its callback routine from the list of registered callbacks and to remove its reference to the callback object.

Introduction to Spin Locks

6/25/2019 • 2 minutes to read • Edit Online

Spin locks are kernel-defined, kernel-mode-only synchronization mechanisms, exported as an opaque type: KSPIN_LOCK. A spin lock can be used to protect shared data or resources from simultaneous access by routines that can execute concurrently and at IRQL >= DISPATCH_LEVEL in SMP machines.

Many components use spin locks, including drivers. Any kind of driver might use one or more *executive spin locks*. For example, most file systems use an interlocked work queue in the file system driver's (FSD's) device extension to store IRPs that are processed both by the file system's worker-thread callback routines and by the FSD. An interlocked work queue is protected by an executive spin lock, which resolves contention among the FSD trying to insert IRPs into the queue and any threads simultaneously trying to remove IRPs. As another example, the system floppy controller driver uses two executive spin locks. One executive spin lock protects an interlocked work queue shared with this driver's device-dedicated thread; the other protects a timer object shared by three driver routines.

Drivers for Microsoft Windows XP and later versions of Windows can use **KeAcquireInStackQueuedSpinLock** and **KeReleaseInStackQueuedSpinLock** to acquire and release the spin lock as a *queued spin lock*. Queued spin locks provide better performance than ordinary spin locks for high contention locks on multiprocessor machines. For more information, see Queued Spin Locks. Drivers for Windows 2000 can use **KeAcquireSpinLock** and **KeReleaseSpinLock** to acquire and release a spin lock as an ordinary spin lock.

To synchronize access to simple data structures, drivers can use any of the **ExInterlocked*Xxx*** routines to ensure atomic access to the data structure. Drivers that use these routines do not need to acquire or release the spin lock explicitly.

Every driver that has an ISR uses an *interrupt spin lock* to protect any data or hardware shared between its ISR and its *SynchCritSection* routines that are usually called from its *StartIo* and *DpcForIsr* routines. An interrupt spin lock is associated with the set of interrupt objects created when the driver calls **IoConnectInterrupt**, as described in Registering an ISR.

Follow these guidelines for using spin locks in drivers:

- Provide the storage for any data or resource protected by a spin lock and for the corresponding spin lock in resident system-space memory (nonpaged pool, as shown in the Virtual Memory Spaces and Physical Memory figure). A driver must provide the storage for any executive spin locks it uses. However, a device driver need not provide the storage for an interrupt spin lock unless it has a multivector ISR or has more than one ISR, as described in Registering an ISR.
- Call **KeInitializeSpinLock** to initialize each spin lock for which the driver provides storage before using it to synchronize access to the shared data or resource it protects.
- Call every support routine that uses a spin lock at an appropriate IRQL, generally at <= DISPATCH_LEVEL for executive spin locks or at <= DIRQL for an interrupt spin lock associated with the driver's interrupt objects.
- Implement routines to execute as quickly as possible while they hold a spin lock. No routine should hold a spin lock for longer than 25 microseconds.
- Never implement routines that do any of the following while holding a spin lock:
  - Cause hardware exceptions or raise software exceptions.
  - Attempt to access pageable memory.

- Make a recursive call that would cause a deadlock or could cause a spin lock to be held for longer than 25 microseconds.
- Attempt to acquire another spin lock if doing so might cause a deadlock.
- Call an external routine that violates any of the preceding rules.

Providing Storage for Spin Locks and Protected Data

6/25/2019 • 2 minutes to read • Edit Online

As part of device start-up, a driver must allocate resident storage for any spin-lock-protected data or resources and for corresponding spin locks in one of the following places:

- The device extension of a device object that the driver sets up by calling **IoCreateDevice**
- The controller extension of a controller object that the driver sets up by calling **IoCreateController**
- Nonpaged, system-space memory that the driver obtains by calling **ExAllocatePoolWithTag**

Attempting to access pageable data while holding a spin lock causes a fatal page fault if that page is not present. Referencing a spin lock that is invalid (because it was stored in pageable memory and is currently paged out) also causes a fatal page fault.

A driver must provide the storage for each of the following kinds of executive spin lock it might use:

- Any spin lock that the driver explicitly acquires and releases using any of the **Ke*Xxx*** spin lock routines.
- Any spin lock used as a parameter to any of the **ExInterlocked*Xxx*** routines.

While a driver can make calls to the **ExInterlocked*Xxx*** routines from its ISR or *SynchCritSection* routines, it cannot use any of the **Ke*Xxx*** routines to acquire and release spin locks at any IRQL greater than DISPATCH_LEVEL. Consequently, any driver that reuses a spin lock between calls to the **Ke*Xxx*SpinLock** and **ExInterlocked*Xxx*** routines must make every call while running at IRQL <= DISPATCH_LEVEL.

A driver can pass the same spin lock to **ExInterlockedInsertHeadList** as it does to another **ExInterlocked*Xxx*** routine, as long as both routines use the spin lock at the same IRQL. For more information about how spin lock usage affects performance, see Using Spin Locks: An Example.

In addition to the storage for its executive spin locks, a device driver must provide the storage for another spin lock to be associated with its interrupt objects if it has a multivector ISR or more than one ISR.

Initializing Spin Locks

6/25/2019 • 2 minutes to read • Edit Online

Before calling any support routine that requires access to a caller-supplied executive spin lock, a driver must call **KeInitializeSpinLock** to initialize the corresponding executive spin lock. Support routines that require an initialized executive spin lock include the following:

- **KeAcquireSpinLock** and, subsequently, **KeReleaseSpinLock**
- **KeAcquireSpinLockAtDpcLevel** and, subsequently, **KeReleaseSpinLockFromDpcLevel**
- **KeAcquireInStackQueuedSpinLock** and, subsequently, **KeReleaseInStackQueuedSpinLock**
- **KeAcquireInStackQueuedSpinLockAtDpcLevel** and, subsequently, **KeReleaseInStackQueuedSpinLockFromDpcLevel**
- An **ExInterlocked*Xxx*** routine

Before calling **IoConnectInterrupt** and **KeSynchronizeExecution**, a lowest-level driver must call **KeInitializeSpinLock** to initialize an interrupt spin lock for which it provides storage.

Calling Support Routines That Use Spin Locks

6/25/2019 • 2 minutes to read • Edit Online

Calling **KeAcquireSpinLock** or **KeAcquireInStackQueuedSpinLock** sets the IRQL on the current processor to DISPATCH_LEVEL until a corresponding call to **KeReleaseSpinLock** or **KeReleaseInStackQueuedSpinLock** restores the previous IRQL. Consequently, drivers must be executing at IRQL <= DISPATCH_LEVEL when they call **KeAcquireSpinLock** or **KeAcquireInStackQueuedSpinLock**.

Callers of **KeAcquireSpinLockAtDpcLevel**, **KeAcquireInStackQueuedSpinLockAtDpcLevel**, **KeReleaseInStackQueuedSpinLockFromDpcLevel**, and **KeReleaseSpinLockFromDpcLevel** run faster because they are already running at IRQL = DISPATCH_LEVEL so these support routines need not reset IRQL on the current processor. Consequently, it is a fatal error on most Windows platforms to call **KeAcquireSpinLockAtDpcLevel** or **KeAcquireInStackQueuedSpinLockAtDpcLevel** while running at IRQL less than DISPATCH_LEVEL. It is also an error to release a spin lock that was acquired with **KeAcquireSpinLock** by calling **KeReleaseSpinLockFromDpcLevel** because the caller's original IRQL is not restored.

Routines that hold an executive spin lock, such as the **ExInterlocked*Xxx***, *usually execute at IRQL = DISPATCH_LEVEL until they release the spin lock and return control to the caller. However, it is possible for a driver's* InterruptService *routine and* SynchCritSection *routines (which run at DIRQL) to call certain* ***ExInterlocked****Xxx routines, such as the* **ExInterlocked*Xxx*List** routines, as long as the spin lock passed to the routine is used exclusively by the ISR and *SynchCritSection* routines.

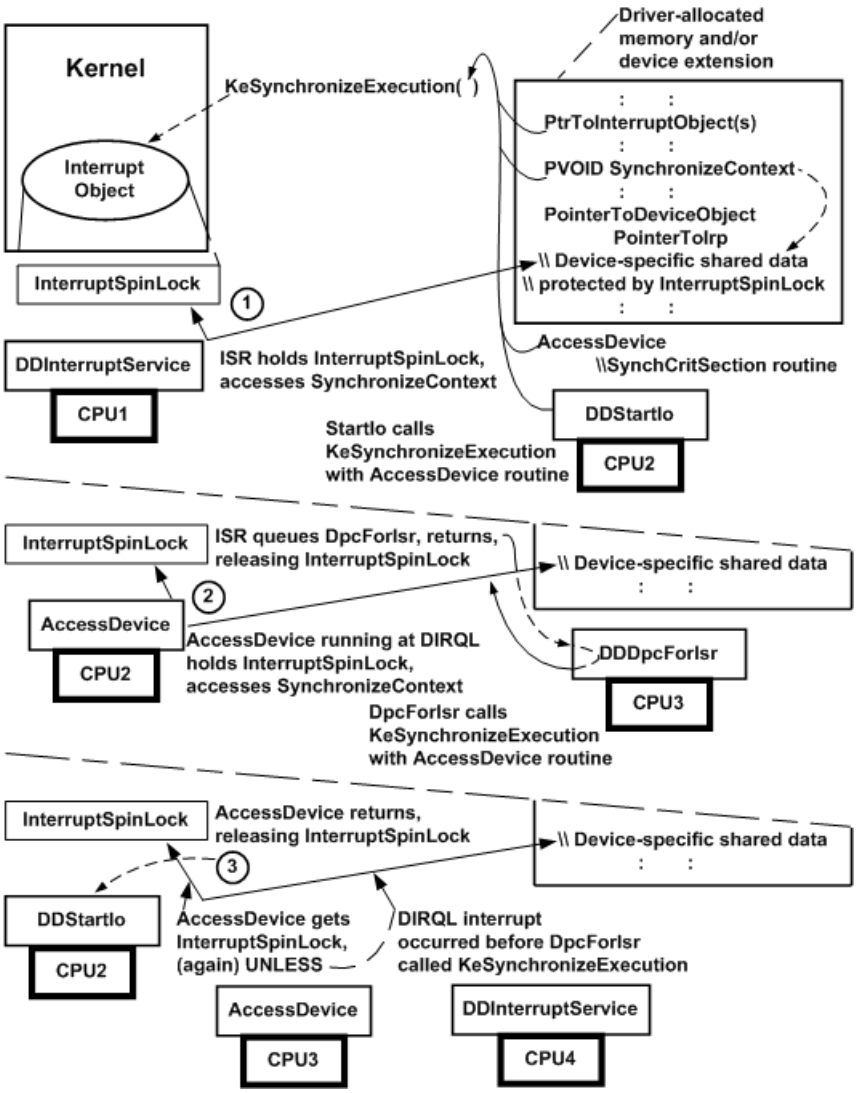
Each routine that holds an interrupt spin lock executes at the DIRQL of an associated set of interrupt objects. Therefore, a driver must not call **KeAcquireSpinLock** and **KeReleaseSpinLock** nor any other routine that uses an executive spin lock from its ISR or *SynchCritSection* routines. Such a call is an error that can cause a system deadlock, requiring the user to reboot his or her machine. Note that if a driver's ISR or *SynchCritSection* routine calls an **ExInterlocked*Xxx*List** routine, the driver cannot reuse the spin lock it passes to the **ExInterlocked*Xxx*List** routines in calls to the **Ke*Xxx*SpinLock** or **Ke*Xxx*SpinLock*Xxx*DpcLevel** support routines.

If a driver has a multivector ISR or more than one ISR, its routines can call **KeSynchronizeExecution** while executing at any IRQL up to the *SynchronizeIrql* value specified for the associated interrupt objects when they were connected.

Using Spin Locks: An Example

6/25/2019 • 4 minutes to read • Edit Online

Minimizing the time that a driver holds spin locks can significantly improve both the performance of the driver and of the system overall. For example, consider the following figure, which shows how an interrupt spin lock protects device-specific data that must be shared between an ISR and the *StartIo* and *DpcForIsr* routines in an SMP machine.



1. While the driver's ISR runs at DIRQL on one processor, its *StartIo* routine runs at DISPATCH_LEVEL on a second processor. The kernel interrupt handler holds the InterruptSpinLock for the driver's ISR, which accesses protected, device-specific data, such as state or pointers to device registers (SynchronizeContext), in the driver's device extension. The *StartIo* routine, which is ready to access SynchronizeContext, calls **KeSynchronizeExecution**, passing a pointer to the associated interrupt objects, the shared SynchronizeContext, and the driver's *SynchCritSection* routine (AccessDevice in the previous figure).

   Until the ISR returns, thereby releasing the driver's InterruptSpinLock, **KeSynchronizeExecution***spins* on the second processor, preventing AccessDevice from touching SynchronizeContext. However, **KeSynchronizeExecution** also raises IRQL on the second processor to the SynchronizeIrql of the interrupt objects, thereby preventing another device interrupt from occurring on that processor so that AccessDevice can be run at DIRQL as soon as the ISR returns. However, higher DIRQL interrupts for other devices, clock interrupts, and power-fail interrupts can still occur on either processor.

2. When the ISR queues the driver's *DpcForIsr* and returns, AccessDevice runs on the second processor at the SynchronizeIrql of the associated interrupt objects and accesses SynchronizeContext. Meanwhile, the *DpcForIsr* is run on another processor at DISPATCH_LEVEL IRQL. The *DpcForIsr* also is ready to access SynchronizeContext, so it calls **KeSynchronizeExecution** using the same parameters that the *StartIo* routine did in Step 1.

   When **KeSynchronizeExecution** acquires the spin lock and runs AccessDevice on behalf of the *StartIo* routine, the driver-supplied synchronization routine AccessDevice is given exclusive access to SynchronizeContext. Because AccessDevice runs at the SynchronizeIrql, the driver's ISR cannot acquire the spin lock and access the same area until the spin lock is released, even if the device interrupts on another processor while AccessDevice is executing.

3. AccessDevice returns, releasing the spin lock. The *StartIo* routine resumes running at DISPATCH_LEVEL on the second processor. **KeSynchronizeExecution** now runs AccessDevice on the third processor, so it can access SynchronizeContext on behalf of the *DpcForIsr*. However, if a device interrupt had occurred before the *DpcForIsr* called **KeSynchronizeExecution** in Step 2, the ISR might run on another processor before **KeSynchronizeExecution** could acquire the spin lock and run AccessDevice on the third processor.

As the previous figure shows, while a routine running on one processor holds a spin lock, every other routine trying to acquire that spin lock gets no work done. Each routine trying to acquire an already held spin lock simply spins on its current processor until the holder releases the spin lock. When a spin lock is released, one and only one routine can acquire it; every other routine currently trying to acquire the same spin lock will continue to spin.

The holder of any spin lock runs at a raised IRQL: either at DISPATCH_LEVEL for an executive spin lock, or at a DIRQL for an interrupt spin lock. Callers of **KeAcquireSpinLock** and **KeAcquireInStackQueuedSpinLock** run at DISPATCH_LEVEL until they call **KeReleaseSpinLock** or **KeReleaseInStackQueuedSpinLock** to release the lock. Callers of **KeSynchronizeExecution** automatically raise IRQL on the current processor to the SynchronizeIrql of the interrupt objects until the caller-supplied *SynchCritSection* routine exits and **KeSynchronizeExecution** returns control. For more information, see Calling Support Routines That Use Spin Locks.

Keep in mind the following fact about using spin locks:

All code that runs at a lower IRQL cannot get any work done on the set of processors occupied by a spin-lock holder and by other routines trying to acquire the same spin lock.

Consequently, minimizing the time a driver holds spin locks results in significantly better driver performance and contributes significantly to better overall system performance.

As the previous figure shows, the kernel interrupt handler executes routines running at the same IRQL in a multiprocessor machine on a first-come, first-served basis. The kernel also does the following:

- When a driver routine calls **KeSynchronizeExecution**, the kernel causes the driver's *SynchCritSection* routine to run on the same processor from which the call to **KeSynchronizeExecution** occurred (see Steps 1 and 3).
- When a driver's ISR queues its *DpcForIsr*, the kernel causes the DPC to run on the first available processor on which IRQL falls below DISPATCH_LEVEL. This is not necessarily the same processor from which the **IoRequestDpc** call occurred (see Step 2).

A driver's interrupt-driven I/O operations might tend to be serialized in a uniprocessor machine, but the same operations can be truly asynchronous in an SMP machine. As the previous figure shows, a driver's ISR could run on CPU4 in an SMP machine before its *DpcForIsr* begins processing an IRP for which the ISR has already handled a device interrupt on CPU1.

In other words, you should not assume that an interrupt spin lock can protect operation-specific data that the ISR saves when it runs on one processor from being overwritten by the ISR when a device interrupt occurs on another

processor before the *DpcForIsr* or *CustomDpc* routine runs.

Although a driver could try to serialize all interrupt-driven I/O operations to preserve data collected by the ISR, that driver would not run much faster in an SMP machine than in a uniprocessor machine. To get the best possible driver performance while remaining portable across uniprocessor and multiprocessor platforms, drivers should use some other technique to save operation-specific data obtained by the ISR for subsequent processing by the *DpcForIsr*.

For example, an ISR can save operation-specific data in the IRP it passes to the *DpcForIsr*. A refinement of this technique is to implement a *DpcForIsr* that consults an ISR-augmented count, processes the count's number of IRPs using ISR-supplied data, and resets the count to zero before returning. Of course, the count must be protected by the driver's interrupt spin lock because its ISR and a *SynchCritSection* would maintain its value dynamically.

Preventing Errors and Deadlocks While Using Spin Locks

6/25/2019 • 2 minutes to read • Edit Online

While a driver routine holds a spin lock, it cannot cause a hardware exception or raise a software exception without bringing down the system. In other words, a driver's ISR and any *SynchCritSection* routine that the driver supplies in a call to **KeSynchronizeExecution** must not cause a fault or trap, such as a page fault or an arithmetic exception, and cannot raise a software exception. A routine that calls **KeAcquireSpinLock** or **KeAcquireInStackQueuedSpinLock** also cannot cause a hardware exception or raise a software exception until it has released its executive spin lock and is no longer running at IRQL = DISPATCH_LEVEL.

Pageable Data and Support Routines

While holding a spin lock, drivers must not call routines that access pageable data. Remember that drivers can call certain support routines that access pageable data if and only if their calls occur while executing at an IRQL strictly less than DISPATCH_LEVEL. This IRQL restriction precludes calling these support routines while holding a spin lock. For IRQL requirements for any specific support routine, see the routine's reference page.

Recursion

Attempting to acquire a spin lock recursively is guaranteed to cause a deadlock: the holding instantiation of a recursive routine cannot release the spin lock while a second instantiation spins, trying to acquire the same spin lock.

The following guidelines describe how you use spin locks with recursive routines:

- The recursive routine must not call itself while holding a spin lock, or must not attempt to acquire the same spin lock on subsequent calls.
- While the recursive routine holds a spin lock, another driver routine must not call the recursive routine if recursion might cause a deadlock or could cause the caller to hold the spin lock for longer than 25 microseconds.

For more information about recursive driver routines, see Using the Kernel Stack.

Nested Spin Lock Acquisitions

Attempting to acquire a second spin lock while holding another spin lock also can cause deadlocks or poor driver performance.

The following guidelines describe how drivers should hold spin locks:

- The driver must not call a support routine that uses a spin lock unless a deadlock cannot occur.
- Even if a deadlock cannot occur, the driver should not call a support routine that uses a spin lock unless alternate coding techniques cannot provide comparable driver performance and functionality.
- If a driver makes nested calls to acquire spin locks, it must always acquire the spin locks in the same order each time they are acquired. This order helps avoid deadlocks.

In general, avoid using nested spin locks to protect overlapping subsets or discrete sets of shared data and resources. Consider what can happen if a driver uses two executive spin locks to protect discrete resources, such as a pair of timer objects that might be set individually and collectively by various driver routines. The driver would deadlock intermittently in an SMP machine, whenever either of two routines, each holding one spin lock, tried to acquire the other spin lock.

For more information about acquiring nested spin locks, see Locks, Deadlocks, and Synchronization.

Queued Spin Locks

6/25/2019 • 2 minutes to read • Edit Online

Queued spin locks are a variant of spin locks that are more efficient for high contention locks on multiprocessor machines. On multiprocessor machines, using queued spin locks guarantees that processors acquire the spin lock on a first-come first-served basis. Drivers for Windows XP and later versions of Windows should use queued spin locks instead of ordinary spin locks.

The driver supplies storage for the spin lock, and initializes it with **KeInitializeSpinLock**. The driver uses **KeAcquireInStackQueuedSpinLock** to acquire the queued spin lock, and **KeReleaseInStackQueuedSpinLock** to release it.

The driver allocates a **KLOCK_QUEUE_HANDLE** structure that it passes by pointer to **KeAcquireInStackQueuedSpinLock**. The driver passes the same structure by pointer to **KeReleaseInStackQueuedSpinLock** when it releases the spin lock. Drivers should normally allocate the structure on the stack each time they acquire the lock.

Drivers must not mix calls to the queued spin lock routines and the ordinary **Ke*Xxx*SpinLock** routines on the same spin lock.

If the driver is already at IRQL = DISPATCH_LEVEL, it can call **KeAcquireInStackQueuedSpinLockAtDpcLevel** and **KeReleaseInStackQueuedSpinLockFromDpcLevel** instead.

Reader/Writer Spin Locks

6/25/2019 • 2 minutes to read • Edit Online

Starting with Windows Vista with Service Pack 1 (SP1), a set of related routines use spin locks to support synchronized access to data structures that are shared by readers and writers. A thread that requires only read access to a data structure can use a spin lock to share this structure with other reader threads. A thread that needs to write to a shared data structure must use the spin lock to obtain exclusive access to the data structure before it can write to this structure.

If a reader thread needs to acquire a spin lock for shared access, and the lock is already held for exclusive access by a writer thread, the reader must first wait for the writer to release the lock. Similarly, if a writer thread needs to acquire a spin lock for exclusive access, and the lock is already held for shared access by one or more reader threads, the writer thread must wait for all of these reader threads to release the lock. While the writer waits, no new reader threads can acquire the lock. Instead, a reader that needs to acquire the lock that the writer is waiting for must first wait for the writer to acquire and release the lock.

A thread can switch roles between reader and writer. A thread that already holds a spin lock for shared access can try to convert the access mode of the spin lock from shared mode to exclusive mode. This attempt succeeds if no readers already hold the spin lock for shared access, and if no writer is already waiting to acquire the spin lock for exclusive access.

Recursive acquisition of a spin lock causes deadlock and is not allowed.

The following is a list of the routines that are available to manage reader/writer spin locks starting with Windows Vista with SP1.

ROUTINE NAME	DESCRIPTION
ExAcquireSpinLockExclusive	Acquires a spin lock for exclusive access by the caller, and raises the IRQL to DISPATCH_LEVEL.
ExAcquireSpinLockExclusiveAtDpcLevel	Acquires a spin lock for exclusive access by a caller that is already running at IRQL >= DISPATCH_LEVEL.
ExAcquireSpinLockShared	Acquires a spin lock for shared access by the caller, and raises the IRQL to DISPATCH_LEVEL.
ExAcquireSpinLockSharedAtDpcLevel	Acquires a spin lock for shared access by a caller that is already running at IRQL >= DISPATCH_LEVEL.
ExReleaseSpinLockExclusive	Releases a spin lock that the caller acquired for exclusive access, and restores the original IRQL.
ExReleaseSpinLockExclusiveFromDpcLevel	Releases a spin lock that the caller acquired for exclusive access, and does not lower the IRQL.
ExReleaseSpinLockShared	Releases a spin lock that the caller acquired for shared access, and restores the original IRQL.
ExReleaseSpinLockSharedFromDpcLevel	Releases a spin lock that the caller acquired for shared access, and does not lower the IRQL.

ROUTINE NAME	DESCRIPTION
ExTryConvertSharedSpinLockExclusive	Tries to convert the access state of a spin lock that the caller already holds for shared access to exclusive access.

The reader/writer spin lock routines all take, as their first parameter, a pointer to a spin lock, which is an **EX_SPIN_LOCK** structure. This structure is opaque to drivers. A driver should allocate the storage for the spin lock from nonpaged system memory, and initialize the lock to zero.

Fast Mutexes and Guarded Mutexes

6/25/2019 • 4 minutes to read • Edit Online

Starting with Windows 2000, drivers can use *fast mutexes* if they require a low-overhead form of mutual exclusion for code that runs at IRQL <= APC_LEVEL. A fast mutex can protect a code path that must be entered by only one thread at a time. To enter the protected code path, the thread *acquires* the mutex. If another thread has already acquired the mutex, execution of the current thread is suspended until the mutex is released. To exit the protected code path, the thread *releases* the mutex.

Starting with Windows Server 2003, drivers can also use *guarded mutexes*. Guarded mutexes are drop-in replacements for fast mutexes but provide better performance. Like a fast mutex, a guarded mutex can protect a code path that must be entered by only one thread at a time. However, code that uses guarded mutexes runs more quickly than code that uses fast mutexes.

In versions of Windows before Windows 8, guarded mutexes are implemented differently from fast mutexes. A code path that is protected by a fast mutex runs at IRQL = APC_LEVEL. A code path that is protected by a guarded mutex runs at IRQL <= APC_LEVEL but with all APCs disabled. In these earlier versions of Windows, acquisition of a guarded mutex is a faster operation than acquisition of a fast mutex. However, these two types of mutex behave identically and are subject to the same restrictions. In particular, kernel routines that are illegal to call at IRQL = APC_LEVEL should not be called from a code path that is protected by either a fast mutex or a guarded mutex.

Starting with Windows 8, guarded mutexes are implemented as fast mutexes. In a code path that is protected by a guarded mutex or a fast mutex, Driver Verifier treats calls to kernel routines as occurring at IRQL = APC_LEVEL. As in earlier versions of Windows, calls that are illegal at APC_LEVEL are illegal in a code path that is protected by a guarded mutex or a fast mutex.

Fast Mutexes

A fast mutex is represented by a **FAST_MUTEX** structure. The driver allocates its own storage for a **FAST_MUTEX** structure and then calls the **ExInitializeFastMutex** routine to initialize the structure.

A thread acquires a fast mutex by doing one of the following:

- Calling the **ExAcquireFastMutex** routine. If the mutex has already been acquired by another thread, execution of the calling thread is suspended until the mutex becomes available.
- Calling the **ExTryToAcquireFastMutex** routine to try to acquire the fast mutex without suspending the current thread. The routine returns immediately, regardless of whether the mutex has been acquired. **ExTryToAcquireFastMutex** returns **TRUE** if it successfully acquired the mutex for the caller; otherwise, it returns **FALSE**.

A thread calls **ExReleaseFastMutex** to release a fast mutex that was acquired by either **ExAcquireFastMutex** or **ExTryToAcquireFastMutex**.

A code path that is protected by a fast mutex runs at IRQL = APC_LEVEL. **ExAcquireFastMutex** and **ExTryToAcquireFastMutex** raise the current IRQL to APC_LEVEL, and **ExReleaseFastMutex** restores the original IRQL. Thus, all APCs are disabled while the thread holds a fast mutex.

If a code path is guaranteed to always run at APC_LEVEL, the driver can instead call **ExAcquireFastMutexUnsafe** and **ExReleaseFastMutexUnsafe** to acquire and release a fast mutex. These routines do not change the current IRQL and can be used safely only when the current IRQL is APC_LEVEL.

Fast mutexes cannot be acquired recursively. If a thread that is already holding a fast mutex tries to acquire it, that

thread will deadlock. Fast mutexes can be used only in code that runs at IRQL <= APC_LEVEL.

Guarded Mutexes

Guarded mutexes, which are available starting with Windows Server 2003, perform the same function as fast mutexes but with higher performance.

Starting with Windows 8, guarded mutexes and fast mutexes are implemented identically.

In versions of Windows before Windows 8, guarded mutexes are implemented differently from fast mutexes. Acquiring a fast mutex raises the current IRQL to APC_LEVEL, while acquiring a guarded mutex enters a guarded region, which is a faster operation. For more information about guarded regions, see Critical Regions and Guarded Regions.

A guarded mutex is represented by a **KGUARDED_MUTEX** structure. The driver allocates its own storage for a **KGUARDED_MUTEX** structure and then calls the **KeInitializeGuardedMutex** routine to initialize the structure.

A thread acquires a guarded mutex by doing one of the following:

- Calling **KeAcquireGuardedMutex**. If the mutex has already been acquired by another thread, execution of the calling thread is suspended until the mutex becomes available.
- Calling **KeTryToAcquireGuardedMutex** to try to acquire the guarded mutex without suspending the current thread. The routine returns immediately, regardless of whether the mutex has been acquired. **KeTryToAcquireGuardedMutex** returns **TRUE** if it successfully acquired the mutex for the caller; otherwise, it returns **FALSE**.

A thread calls **KeReleaseGuardedMutex** to release a guarded mutex that was acquired by either **KeAcquireGuardedMutex** or **KeTryToAcquireGuardedMutex**.

A thread that holds a guarded mutex implicitly runs inside a guarded region. **KeAcquireGuardedMutex** and **KeTryToAcquireGuardedMutex** enter the guarded region, while **KeReleaseGuardedMutex** exits it. All APCs are disabled while the thread holds a guarded mutex.

If a code path is guaranteed to run with all APCs disabled, the driver can instead use **KeAcquireGuardedMutexUnsafe** and **KeReleaseGuardedMutexUnsafe** to acquire and release the guarded mutex. These routines do not enter or exit a guarded region and can be used only inside an already-existing guarded region or at IRQL = APC_LEVEL.

Guarded mutexes cannot be acquired recursively. If a thread that is already holding a guarded mutex tries to acquire it, that thread will deadlock. Guarded mutexes can be used only in code that runs at IRQL <= APC_LEVEL.

ERESOURCE Structures

12/5/2018 • 2 minutes to read • Edit Online

You can use the ERESOURCE structures to implement read/writer locking in your driver. The system provides a set of routines to manipulate the ERESOURCE structures, which are documented in this section.

This section contains the following topic:

Introduction to ERESOURCE Routines

6/25/2019 • 2 minutes to read • Edit Online

The system provides routines to acquire and release ERESOURCE structures, as well as to examine their current state.

Acquiring and Releasing an ERESOURCE Structure

Drivers can use the ERESOURCE structures to implement *exclusive/shared synchronization*. Exclusive/shared synchronization works as follows:

- Any number of threads can acquire an ERESOURCE as shared.
- Only one thread can acquire an ERESOURCE exclusively. The ERESOURCE can only be acquired exclusively if no threads have already acquired it as shared.

A thread that cannot currently acquire an ERESOURCE can optionally be put in a wait state until the ERESOURCE can be acquired. The system maintains two lists of threads that are waiting for an ERESOURCE: a list of *exclusive waiters* and a list of *shared waiters*.

A typical use for exclusive/shared synchronization is to implement a read/write lock. A read/write lock allows several threads to perform a read operation, but only one thread can write at a time. This can be implemented directly in terms of acquiring an ERESOURCE.

A driver allocates the storage for an ERESOURCE and initializes it with **ExInitializeResourceLite**. The system maintains a list of all ERESOURCE structures in use. When the driver no longer requires a particular ERESOURCE, it must call **ExDeleteResourceLite** to delete it from the system's list. The driver can also reuse an ERESOURCE by calling **ExReinitializeResourceLite**.

Drivers can perform the following basic operations on an ERESOURCE:

- Acquire an ERESOURCE as shared with **ExAcquireResourceSharedLite**. This routine acquires the resource only if the resource has not been acquired exclusively and there are no exclusive waiters.
- Acquire an ERESOURCE exclusively with **ExAcquireResourceExclusiveLite**. This routine acquires the resource as long as it has not been acquired either exclusively or as shared.
- Convert an exclusive acquisition to a shared acquisition with **ExConvertExclusiveToSharedLite**.
- Release an acquired resource with **ExReleaseResourceLite**.

The *Wait* parameter of **ExAcquireResourceSharedLite** and **ExAcquireResourceExclusiveLite** determines whether the current thread waits for the ERESOURCE to be acquired. If you specify a value of **FALSE** and the ERESOURCE cannot be acquired, then the routine returns **FALSE**. If you specify a value of **TRUE**, then the current thread is put on the appropriate wait list for the ERESOURCE.

Examining the State of an ERESOURCE Structure

A driver can also determine the current state of an ERESOURCE, as follows:

- Use **ExIsResourceAcquiredLite** or **ExIsResourceAcquiredSharedLite** to determine if the ERESOURCE has already been acquired as either shared or exclusive. Use **ExIsResourceAcquiredExclusiveLite** to check whether the ERESOURCE has been specifically acquired exclusively.
- Use **ExGetSharedWaiterCount** to determine the number of shared waiters for the ERESOURCE, and use **ExGetExclusiveWaiterCount** to determine the number of exclusive waiters for the ERESOURCE.

IoTimer Routines

6/25/2019 • 2 minutes to read • Edit Online

Drivers that need to be called periodically to determine if a device operation has timed out, to update some driver-defined variable (such as a counter), or to time any operation for which small time intervals are not required, can use an *IoTimer* routine. An *IoTimer* routine is actually a DPC routine, associated with a device object, that the I/O manager calls once per second. A driver can have an *IoTimer* routine for each device object that it creates.

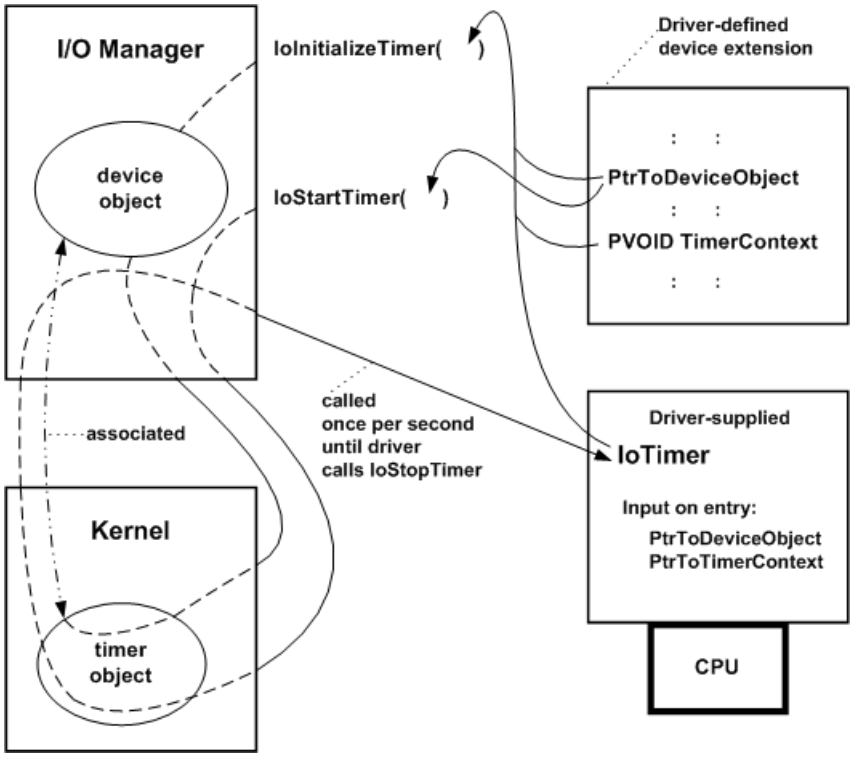
In general, a driver should use an *IoTimer* routine to time operations that require regular one-second intervals. To time operations that require variable intervals or intervals shorter than once per second, a driver should allocate a timer object. For more information, see Timer Objects and DPCs.

This section contains the following topics:

Registering and Enabling an IoTimer Routine

6/25/2019 • 2 minutes to read • Edit Online

Any driver can register an *IoTimer* routine, after it creates one or more device objects, by calling **IoInitializeTimer**. The driver can then start the timer by calling **IoStartTimer**. The following figure illustrates these calls.



After calling **IoCreateDevice** to create device objects, a driver can call **IoInitializeTimer** with the entry point of its *IoTimer* routine, along with pointers to a driver-created device object and a context area in which the driver maintains whatever context its *IoTimer* routine uses. The I/O manager associates the device object with a kernel-allocated timer object and sets up the timer object to time out every second.

After the driver calls **IoStartTimer**, its *IoTimer* routine is called once per second until the driver calls **IoStopTimer**. A driver can reenable calls to its *IoTimer* routine with **IoStartTimer**. (Frequently, when a driver calls **IoStartTimer**, it supplies the device object pointer obtained from the I/O stack location of the current IRP.)

On entry, the *IoTimer* routine is passed the device object pointer, along with the context pointer that the driver supplied when it called **IoInitializeTimer**.

Drivers must not call **IoStopTimer** from within an *IoTimer* routine.

The I/O manager unregisters the timer routine for a specified device object and frees its associated context when the driver calls **IoDeleteDevice**.

Providing IoTimer Context Information

6/25/2019 • 2 minutes to read • Edit Online

The *Context* pointer passed to **IoInitializeTimer** identifies a context area where other driver routines, and the *IoTimer* routine itself, can maintain state about timed operations. The I/O manager passes the *Context* pointer whenever it calls the *IoTimer* routine.

Because an *IoTimer* routine is run at IRQL = DISPATCH_LEVEL, its context area must be in resident, system-space memory. Most drivers that have *IoTimer* routines use the device extension of the associated device object as a *Context*-accessible area, but the context can instead be in a controller extension if the driver uses a controller object or in nonpaged pool allocated by the driver.

Follow these guidelines for an *IoTimer*routine's context area:

- If the *IoTimer* routine shares its context area with the driver's ISR, it must use **KeSynchronizeExecution** to call a *SynchCritSection* routine that accesses the context area in a multiprocessor-safe manner. For more information, see Using Critical Sections.
- If the *IoTimer* routine does not share its context area with an ISR, but does share it with another driver routine, the driver must protect the shared context area with an initialized executive spin lock, in order to access the context information in a multiprocessor-safe manner. For more information, see Spin Locks.

Using an IoTimer Routine

6/25/2019 • 3 minutes to read • Edit Online

While the timer for the associated device object is enabled, the *IoTimer* routine is called approximately once per second. However, because the intervals at which each *IoTimer* routine is called depend on the resolution of the system clock, do not assume that an *IoTimer* routine will be called precisely on a one-second boundary.

Note An *IoTimer* routine, like all DPC routines, is called at IRQL = DISPATCH_LEVEL. While a DPC routine runs, all threads are prevented from running on the same processor. Driver developers should carefully design their *IoTimer* routines to run for as brief a time as possible.

Perhaps the most common use for an *IoTimer* routine is to time out device I/O operations for an IRP. Consider the following scenario for using an *IoTimer* routine as a running timer within a device driver:

1. When it starts the device, the driver initializes a timer counter in the device extension to -1, indicating no current device I/O operations, and calls **IoStartTimer** just before it returns STATUS_SUCCESS.

   Each time the *IoTimer* routine is called, it checks whether the timer counter is -1, and, if so, returns control.

2. The driver's *StartIo* routine initializes the timer counter in the device extension to an upper limit, plus an additional second in case the *IoTimer* routine has just been run. It then uses **KeSynchronizeExecution** to call a *SynchCritSection_1* routine, which programs the physical device for the operation requested by the current IRP.

3. The driver's ISR resets the timer counter to -1 before queuing the driver's *DpcForIsr* routine or a *CustomDpc* routine.

4. Each time the *IoTimer* routine is called, it checks whether the timer counter has been reset by the ISR to -1, and, if so, returns control. If not, the *IoTimer* routine uses **KeSynchronizeExecution** to call a *SynchCritSection_2* routine, which adjusts the timer counter by some driver-determined number of seconds.

5. The *SynchCritSection_2* routine returns **TRUE** to the *IoTimer* routine as long as the current request has not yet timed out. If the timer counter goes to zero, the *SynchCritSection_2* routine resets the timer counter to a driver-determined reset-timeout value, sets a reset-expected flag for itself (and for the *DpcForIsr*) in its context area, attempts to reset the device, and returns **TRUE**.

   The *SynchCritSection_2* routine will be called again if its reset operation also times out on the device, when it returns **FALSE**. If its reset succeeds, the *DpcForIsr* routine determines that the device has been reset from the reset-expected flag and retries the request, repeating the actions of the *StartIo* routine as described in Step 2.

6. If the *SynchCritSection_2* routine returns **FALSE**, the *IoTimer* routine assumes the physical device is in an unknown state because an attempt to reset it has already failed. In these circumstances, the *IoTimer* routine queues a *CustomDpc* routine and returns. This *CustomDpc* routine logs a device I/O error, calls **IoStartNextPacket**, fails the current IRP, and returns.

If this device driver's ISR resets the shared timer counter to -1, as described in Step 3, the driver's *DpcForIsr* routine completes the interrupt-driven I/O processing of the current IRP. The reset timer counter indicates that this device I/O operation has not timed out, so the *IoTimer* routine does not need to change the timer counter.

Under most circumstances, the preceding *SynchCritSection_2* routine simply decrements the timer counter. The *SynchCritSection_2* routine attempts to reset the device only if the current I/O operation has timed out, which is indicated when the timer counter goes to zero. And only if an attempt to reset the device has already failed does the *SynchCritSection_2* routine return **FALSE** to the *IoTimer* routine.

Consequently, both the preceding *IoTimer* routine and its helper *SynchCritSection_2* routine take very little time to execute under normal circumstances. By using an *IoTimer* routine in this manner, a device driver ensures that each valid device I/O request can be retried, if necessary, and that a *CustomDpc* routine will fail an IRP only if an uncorrectable hardware failure prevents the IRP from being satisfied. Moreover, the driver provides this functionality at very little cost in execution time.

The simplicity of the preceding scenario depends on a device that does only one operation at a time and on a driver that does not normally overlap I/O operations. A driver that carries out overlapped device I/O operations, or a higher-level driver that uses an *IoTimer* routine to time out a set of driver-allocated IRPs sent to more than one chain of lower drivers, would have more complex timeout scenarios to manage.

Counters

6/25/2019 • 2 minutes to read • Edit Online

The system provides several driver support routines that return various count values.

KeQuerySystemTime

KeQueryInterruptTime

KeQueryInterruptTimePrecise

KeQueryTickCount

KeQueryPerformanceCounter

KeQueryTimeIncrement

Types of APCs

12/5/2018 • 2 minutes to read • Edit Online

An asynchronous procedure call (APC) is a function that executes asynchronously. APCs are similar to deferred procedure calls (DPCs), but unlike DPCs, APCs execute within the context of a particular thread. Drivers (other than file systems and file-system filter drivers) do not use APCs directly, but other parts of the operating system do, so you need to be aware of how APCs work.

The Windows operating system uses three kinds of APCs:

- *User APCs* run strictly in user mode and only when the current thread is in an alertable wait state. The operating system uses user APCs to implement mechanisms such as overlapped I/O and the **QueueUserApc** Win32 routine.
- *Normal kernel APCs* run in kernel mode at IRQL = PASSIVE_LEVEL. A normal kernel APC preempts all user-mode code, including user APCs. Normal kernel APCs are generally used by file systems and file-system filter drivers.
- *Special kernel APCs* run in kernel mode at IRQL = APC_LEVEL. A special kernel APC preempts user-mode code and kernel-mode code that executes at IRQL = PASSIVE_LEVEL, including both user APCs and normal kernel APCs. The operating system uses special kernel APCs to handle operations such as I/O request completion.

Disabling APCs

6/25/2019 • 2 minutes to read • Edit Online

The system provides three mechanisms to disable APCs for the current thread:

- **Critical regions.** When a thread is inside a critical region, its user APCs and normal kernel APCs are not executed. Special kernel APCs are still executed. For more information about these APC types, see Types of APCs.
- **Guarded regions.** When a thread is inside a guarded region, none of its APCs are executed.
- **Raising the current IRQL to APC_LEVEL or higher.** A thread that is executing at IRQL >= APC_LEVEL executes with all APCs disabled.

Note that these settings apply to the current thread and do not affect the behavior of any other thread.

Some driver support routines must be called with particular kinds of APCs disabled. For example, routines that acquire an executive resource (such as **ExAcquireResourceSharedLite**) must be called with normal kernel APCs disabled. Other routines must be called with particular kinds of APCs enabled. For example, any routine that relies on an I/O completion routine (such as **IoVolumeDeviceToDosName**) must be called with special kernel APCs enabled. The documentation for each routine specifies if the routine has any particular restrictions on the state of APC execution.

A driver can explicitly enter a critical or guarded region by calling the appropriate routine. For more information, see Critical Regions and Guarded Regions. A driver can also explicitly raise the current IRQL to APC_LEVEL by calling **KeRaiseIrql**. The driver must subsequently lower the IRQL to its original value by calling **KeLowerIrql**. Using a guarded region is faster than raising and lowering the current IRQL, but guarded regions are only available in Windows Server 2003 and later versions of Windows.

The following mutex operations have the same effect as entering or leaving a critical or guarded region or raising or lowering the current IRQL:

- Holding a mutex object implicitly places the holder within a critical region.
- Holding a guarded mutex implicitly places the holder within a guarded region.
- Holding a fast mutex implicitly raises the current IRQL to APC_LEVEL.

For more information about mutex objects, see Mutex Objects. For more information about fast and guarded mutexes, see Fast Mutexes and Guarded Mutexes.

Critical Regions and Guarded Regions

6/25/2019 • 2 minutes to read • Edit Online

A thread that is inside a *critical region* executes with user APCs and normal kernel APCs disabled. A thread inside a *guarded region* runs with all APCs disabled.

Critical Regions

A driver can enter and exit a critical region as follows:

- Call **KeEnterCriticalRegion** to enter a critical region.
- Call **KeLeaveCriticalRegion** to exit a critical region.

Each call to **KeEnterCriticalRegion** must have a matching call to **KeLeaveCriticalRegion**.

Guarded Regions

A driver can enter and exit a guarded region as follows:

- Call **KeEnterGuardedRegion** to enter a guarded region.
- Call **KeLeaveGuardedRegion** to leave a guarded region.

Each call to **KeEnterGuardedRegion** must have a matching call to **KeLeaveGuardedRegion**.

Drivers that were developed for Windows Server 2003 and later versions of Windows can use guarded regions to disable special kernel APCs. Drivers that were developed for earlier operating systems can disable special kernel APCs by raising the current IRQL to APC_LEVEL by calling **KeRaiseIrql**. Use **KeLowerIrql** to lower the current IRQL to the previous value.

Acquire and Release Semantics

6/25/2019 • 2 minutes to read • Edit Online

An operation has *acquire semantics* if other processors will always see its effect before any subsequent operation's effect. An operation has *release semantics* if other processors will see every preceding operation's effect before the effect of the operation itself.

Consider the following code example:

```
a++;
b++;
c++;
```

From another processor's point of view, the preceding operations can appear to occur in any order. For example, the other processor might see the increment of `b` before the increment of `a`.

Atomic operations, such as those that the **InterlockedXxx** routines perform, have both acquire and release semantics by default. However, Itanium-based processors execute operations that have only acquire or only release semantics faster than those that have both. Therefore, the system provides **InterlockedXxxAcquire** and **InterlockedXxxRelease** versions of some of the **InterlockedXxx** routines.

For example, the **InterlockedIncrementAcquire** routine uses acquire semantics to increment a variable. If you rewrote the preceding code example as follows:

```
InterlockedIncrementAcquire(&a);
b++;
c++;
```

other processors would always see the increment of `a` before the increments of `b` and `c`.

Likewise, the **InterlockedIncrementRelease** routine uses release semantics to increment a variable. If you rewrote the code example once again, as follows:

```
a++;
b++;
InterlockedIncrementRelease(&c);
```

other processors would always see the increments of `a` and `b` before the increment of `c`.

If the processor does not provide instructions that have only acquire or only release semantics, the system will use the corresponding routine that provides both types of semantics. For example, on x86 processors both **InterlockedIncrementAcquire** and **InterlockedIncrementRelease** are equivalent to **InterlockedIncrement**.

The following table lists the routines that have acquire-only and release-only variants.

INTERLOCKED*XXX* ROUTINE	ACQUIRE-SEMANTICS-ONLY VERSION	RELEASE-SEMANTICS-ONLY VERSION
InterlockedIncrement	InterlockedIncrementAcquire	InterlockedIncrementRelease
InterlockedDecrement	InterlockedDecrementAcquire	InterlockedDecrementRelease

INTERLOCKED*XXX* ROUTINE	ACQUIRE-SEMANTICS-ONLY VERSION	RELEASE-SEMANTICS-ONLY VERSION
InterlockedCompareExchange	InterlockedCompareExchangeAcquire	InterlockedCompareExchangeRelease

Run-Down Protection

6/25/2019 • 5 minutes to read • Edit Online

Starting with Windows XP, run-down protection is available to kernel-mode drivers. Drivers can use run-down protection to safely access objects in shared system memory that are created and deleted by another kernel-mode driver.

An object is said to be *run down* if all outstanding accesses of the object are finished and no new requests to access the object will be granted. For example, a shared object might need to be run down so that it can be deleted and replaced with a new object.

The driver that owns the shared object can enable other drivers to acquire and release run-down protection on the object. When run-down protection is in effect, a driver other than the owner can access the object without risk that the owner will delete the object before the access completes. Before the access starts, the accessing driver requests run-down protection on the object. For a long-lived object, this request is nearly always granted. After the access finishes, the accessing driver releases its previously acquired run-down protection on the object.

Primary run-down protection routines

To start sharing an object, the driver that owns the object calls the **ExInitializeRundownProtection** routine to initialize run-down protection on the object. After this call, other drivers that access the object can acquire and release run-down protection on the object.

A driver that accesses the shared object calls the **ExAcquireRundownProtection** routine to request run-down protection on the object. After the access is finished, this driver calls the **ExReleaseRundownProtection** routine to release run-down protection on the object.

If the owning driver determines that the shared object must be deleted, this driver waits to delete the object until all outstanding accesses of the object are finished.

In preparation to delete the shared object, the owning driver calls the **ExWaitForRundownProtectionRelease** routine to wait for the object to run down. During this call, **ExWaitForRundownProtectionRelease** waits for all previously granted instances of run-down protection on the object to be released, but prevents new requests for run-down protection on the object from being granted. After the last protected access finishes and all instances of run-down protection are released, **ExWaitForRundownProtectionRelease** returns, and the owning driver can safely delete the object.

ExWaitForRundownProtectionRelease blocks the execution of the calling driver thread until all drivers that hold run-down protection on the shared object release this protection. To prevent **ExWaitForRundownProtectionRelease** from blocking execution for excessively long periods, drivers threads that access the shared object should avoid being suspended while they hold run-down protection on the object. For this reason, accessing drivers should call **ExAcquireRundownProtection** and **ExReleaseRundownProtection** within a critical region or guarded region, or while running at IRQL = APC_LEVEL.

Uses for run-down protection

Run-down protection is particularly useful for providing access to a shared object that is nearly always available but might occasionally need to be deleted and replaced. Drivers that access data or that call routines in this object must not try to access the object after it is deleted. Otherwise, these invalid accesses might cause unpredictable behavior, data corruption, or even system failure.

For example, an antivirus driver typically stays loaded in memory when the operating system is running. Occasionally, this driver might need to be unloaded and replaced with an updated release of the driver. Other drivers send I/O requests to the antivirus driver to access the data and routines in this driver. Before sending an I/O request, a kernel component, such as a file system filter manager, can acquire run-down protection to guard against premature unloading of the antivirus driver while it handles the I/O request. After the I/O request completes, run-down protection can be released.

Run-down protection does not serialize accesses to a shared object. If two or more accessing drivers can simultaneously hold run-down protection on an object, and accesses to the object must be serialized, some other mechanism, such as a mutual-exclusion lock, must be used to serialize the accesses.

The EX_RUNDOWN_REF structure

An **EX_RUNDOWN_REF** structure tracks the status of run-down protection on a shared object. This structure is opaque to drivers. The system-supplied run-down protection routines use this structure to count the number of instances of run-down protection that are currently in effect on the object. These routines also use this structure to track whether the object is run down or is in the process of being run down.

To start sharing an object, the driver that owns the object calls **ExInitializeRundownProtection** to initialize the **EX_RUNDOWN_REF** structure associated with the object. After initialization, the owning driver can make this structure available to other drivers that require access to the object. The accessing drivers pass this structure as a parameter to the **ExAcquireRundownProtection** and **ExReleaseRundownProtection** calls that acquire and release run-down protection on the object. The owning driver passes this structure as a parameter to the **ExWaitForRundownProtectionRelease** call that waits for the object to run down so that it can be safely deleted.

Comparison to locks

Run-down protection is one of several ways to guarantee safe access to a shared object. Another approach is to use a mutual-exclusion software lock. If a driver requires access to an object that is currently locked by another driver, the first driver must wait for the second driver to release the lock. However, acquiring and releasing locks can become a performance bottleneck, and locks can consume large amounts of memory. If used incorrectly, locks might cause drivers that compete for the same shared objects to become deadlocked. Efforts to detect and avoid deadlocks typically require the diversion of substantial computing resources.

In contrast to locks, run-down protection has relatively lightweight processing time and memory requirements. A simple reference count is associated with the object to ensure that deletion of the object is deferred until all outstanding accesses of the object are completed. With this approach, atomic, interlocked hardware instructions can be used instead of mutual-exclusion software locks to guarantee safe access to an object. Calls to acquire and release run-down protection are typically very fast. The benefits of using a lightweight mechanism, such as run-down protection, can be significant for a shared object that has a long life and is shared among many drivers.

Other run-down protection routines

Several other run-down protection routines are available, in addition to those that were mentioned previously. These additional routines might used by some drivers.

The **ExReInitializeRundownProtection** routine enables a previously used **EX_RUNDOWN_REF** structure to be associated with a new object, and initializes run-down protection on this object.

The **ExRundownCompleted** routine updates the **EX_RUNDOWN_REF** structure to indicate that the run down of the associated object has completed.

The **ExAcquireRundownProtectionEx** and **ExReleaseRundownProtectionEx** routines are similar to **ExAcquireRundownProtection** and **ExReleaseRundownProtection**. These four routines increment or decrement the count of the instances of run-down protection that are in effect on a shared object. Whereas

ExAcquireRundownProtection and **ExReleaseRundownProtection** increment and decrement this count by one, **ExAcquireRundownProtectionEx** and **ExReleaseRundownProtectionEx** increment and decrement the count by arbitrary amounts.

Introduction to the Common Log File System

12/5/2018 • 2 minutes to read • Edit Online

The Common Log File System (CLFS) is a general-purpose logging service that can be used by software *clients* running in user-mode or kernel-mode. This documentation discusses the CLFS interface for kernel-mode clients. For information about the user-mode interface, see Common Log File System in the Microsoft Windows SDK.

CLFS encapsulates all the functionality of the Algorithm for Recovery and Isolation Exploiting Semantics (ARIES). However, the CLFS device driver interface (DDI) is not limited to supporting ARIES; it is well suited to a variety of logging scenarios.

The primary job of any high-performance transactional log is to allow log clients to accurately repeat history. CLFS does this by marshalling client log records into memory buffers, forcing them to stable storage, and reading records back on request. It is important to note that after a record makes it to stable storage and the storage media is intact, CLFS will be able to read the record across system failures.

CLFS supports dedicated logs and multiplexed logs. A dedicated log has a single *stream* of log records that is used by all of the log's clients. A multiplexed log (also called a common log) has several streams. Each stream has its own clients and its own memory buffers for marshalling log records, but the records from all those buffers are multiplexed into a single queue and flushed to a single log on stable storage. Multiplexing allows the I/O operations of several streams to be consolidated.

When a client writes a record to a stream, it gets back a log sequence number (LSN) that identifies the log record for future reference. The LSNs assigned to the records that are written to a particular stream form an increasing sequence. That is, the LSN assigned to a record that is written to a stream is always greater than the LSN assigned to the previous record written to that same stream.

CLFS provides several services in addition to marshalling, flushing, and retrieving log records. The following list describes some of those additional services.

- Space for a set of related log records can be reserved ahead of time. This means that a client can proceed with a transaction knowing that CLFS will be able to append to the log all of the records that describe the transaction.
- CLFS maintains a header for each log record. Clients can set certain fields in the header to create chains of linked records that you can later traverse in reverse order.
- CLFS flushes log records to stable storage according to its policy, but also allows clients to force a set of log records to stable storage.
- CLFS maintains metadata for a log and also for each stream of a multiplexed log. Clients can view metadata and set certain portions of the metadata.
- For each stream, CLFS maintains a base LSN and a last LSN that a client can use to delineate the active portion of the stream.
- For dedicated logs, CLFS maintains (at the client's request) an archive tail that the client can use to keep track of the portion of the log that has been archived.

Certain features of CLFS (for example, the previous LSN and undo-next LSN fields of a record header) can be best understood by studying ARIES. For more information about ARIES, see the following papers.

- C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, Peter Schwarz, *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*.

- C. Mohan, *Repeating History Beyond ARIES*.

CLFS Terminology

12/5/2018 • 3 minutes to read • Edit Online

The following list gives definitions of key terms used in the Common Log File System (CLFS) documentation. These definitions apply during a discussion of CLFS, but might not apply otherwise. Many of these terms have general meanings or meanings in the context of other technologies that differ from the definitions given here.

container
A contiguous extent on a physical disk or other stable storage medium. For example, a container could be a contiguous disk file.

sector
The unit of atomic I/O on a physical storage medium. The size of a sector is a property of a particular storage device. For example, a hard disk might have a sector size of 512 bytes.

log
A base file and a set of logically ordered containers. The base file holds metadata for the log, and the containers hold log records. All the containers are the same size.

client
An application, driver, thread, or other unit of software that uses a CLFS log.

record
The unit of data that a client can append to or read from a log.

stream
An ordered subset of the records in a log. A log can have one or more streams. A client appends records to and reads records from a particular stream. You can compare the records in a given stream to determine the order in which they were written. You cannot compare records in different streams. A given stream can have several clients. For example, several threads could append records to a single stream. To a client, a stream appears as if it were the entire log.

dedicated log
A log that can have only one stream.

multiplexed log
A log that can have several streams.

log I/O block
A buffer where CLFS collects a set of records that are atomically written to stable storage.

marshalling area
A set of log I/O blocks, created, maintained, and scheduled by a CLFS client for gathering log records and writing them to stable storage. The log I/O blocks allocated in volatile memory for a particular marshalling area are all the same size.

Note Even though all the log I/O blocks (in volatile memory) for a particular marshalling area are the same size, the log I/O blocks that are written to stable storage (from that marshalling area) vary in size. For example, if a log I/O block is forced to stable storage before it is full, only the used portion of the block will be written to stable storage.

log sequence number (LSN)
An opaque structure that holds a value that uniquely identifies a log record in a given stream. When a client writes a record to a stream, it gets back an LSN that it can use to identify the record in the future. The LSNs that CLFS

assigns to the records in a stream form an increasing sequence. That is, the LSN assigned to a record in a stream is always greater than the LSN assigned to the record previously written to that same stream.

Note Records across streams are not comparable. That is, you cannot compare the LSNs of two records in different streams to determine which record was written first.

base LSN
The LSN of the oldest record in a stream that is still needed by the stream's clients. The clients are responsible for updating the base LSN.

last LSN
The LSN of the youngest record in a stream that is still needed by the stream's clients. Typically this is the record that was most recently written to the stream, but clients have the option of manually setting the last LSN to point to some earlier record in the stream. Manually setting the last LSN to an earlier record is called *truncating* the stream.

archive tail
The LSN of the oldest record in a log for which archiving has not taken place. Not every log has an archive tail. A log that does not have an archive tail is called *ephemeral*, and a log that has an archive tail is called *non-ephemeral*. When a client specifies that a log has an archive tail, the client is responsible for updating the archive tail.

active portion of a stream
The portion of a stream that is currently in use by its clients. The active portion begins with the record pointed to by the base LSN or the archive tail, whichever is smaller. The active portion ends with the record pointed to by the last LSN.

CLFS Log Sequence Numbers

6/25/2019 • 5 minutes to read • Edit Online

In the Common Log File System (CLFS), each log record in a given stream is uniquely identified by a log sequence number (LSN). When you write a record to a stream, you get back an LSN that identifies that record for future reference.

The LSNs created for a particular stream form a strictly increasing sequence. That is, the LSN assigned to a log record in a given stream is always greater than the LSNs assigned to log records previously written to that same stream. The following functions are available for comparing the LSNs of log records in a given stream.

ClfsLsnNull

ClfsLsnEqual

ClfsLsnGreater

ClfsLsnLess

The constants CLFS_LSN_NULL and CLFS_LSN_INVALID are the lower and upper boundaries for all valid LSNs. Any valid LSN is greater than or equal to CLFS_LSN_NULL. Also, any valid LSN is strictly less than CLFS_LSN_INVALID. Note that CLFS_LSN_NULL is a valid LSN, whereas CLFS_LSN_INVALID is not a valid LSN. Even so, you can compare CLFS_LSN_INVALID to other LSNs by using the functions in the previous list.

For each stream, CLFS keeps track of two special LSNs: the base LSN and the last LSN. Also, each individual log record has two special LSNs (the previous LSN and the undo-next LSN) that you can use to create chains of related log records. The following sections describe these special LSNs in detail.

Base LSN

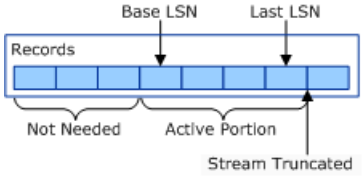
When a client writes the first record in a stream, CLFS sets the base LSN to the LSN of that first record. The base LSN remains unchanged until a client changes it. When the stream's clients no longer need the records prior to a certain point in the stream, they can update the base LSN by calling **ClfsAdvanceLogBase** or **ClfsWriteRestartArea**. For example, if the clients no longer need the first five log records, they can set the base LSN to the LSN of the sixth record.

Last LSN

As clients write records to a stream, CLFS adjusts the last LSN so that it is always the LSN of the last record written. If the clients no longer need the records after a certain point in the stream, they can update the last LSN by calling **ClfsSetEndOfLog**. For example, if the clients no longer need any records written after the tenth record, they can truncate the stream by setting the last LSN to the LSN of the tenth record.

Active portion of a stream

The *active portion* of a stream is the portion of a stream that begins with the record pointed to by the base LSN and ends with the record pointed to by the last LSN. The following diagram illustrates how the base LSN and last LSN delineate the active portion of a stream.



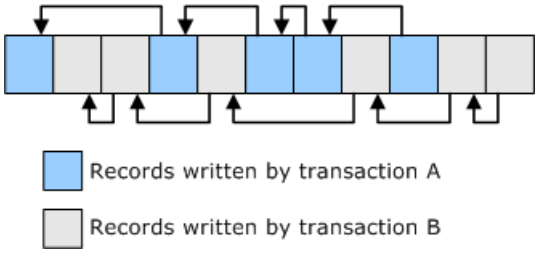
Note If a stream has an archive tail, the active portion of the stream begins at the record pointed to by the base

LSN or the archive tail, whichever is smaller. For more information about archiving, see CLFS Support for Archiving.

Previous LSN

Suppose two active database transactions (transaction A and transaction B) are writing records to the same stream at the same time. Each time transaction A writes a record, it sets the record's previous LSN to the LSN of the previous log record written by transaction A. That forms a chain of log records, belonging to transaction A, that can be traversed in reverse order. The chain ends with the first log record written by transaction A, which has its previous LSN set to CLFS_LSN_INVALID. Similarly, transaction B creates its own chain of log records by setting the previous LSN of each log record it writes.

The arrows in the following diagram illustrate how the previous LSN of a log record points to the previous record in a chain that belongs to a particular transaction.



Undo-next LSN

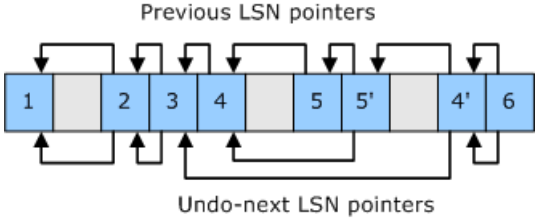
Suppose a transaction makes five updates to a data object in volatile memory, rolls back the fourth and fifth updates, and then makes a sixth update. As the transaction makes the updates, it writes log records 1, 2, 3, 4, 5, 5', 4', and 6. Log records 1 through 5 describe the changes made by updates 1 through 5. Record 5' describes the changes made during the rollback of update 5, and record 4' describes the changes made during the rollback of update 4. Finally, record 6 describes the changes made by update 6. Note that the numbers 1, 2, 3, 4, 5, 5', 4', and 6 are not the LSNs of the log records; they are just numbers used to name the log records for the purpose of this discussion.

Log records 5' and 4', which describe rollbacks, are called compensation log records (CLRs). The transaction sets the undo-next LSN of each CLR to the predecessor (among the records written by the transaction) of the log record whose update was just rolled back (undone). In this example, the undo-next LSN of record 5' is the LSN of record 4, and the undo-next LSN of record 4' is the LSN of record 3.

The ordinary log records (those that are not CLRs), have their undo-next LSNs set to the previous log record written by the transaction. That is, for an ordinary record, the undo-next LSN and previous LSN are the same.

Now suppose there is a system failure and, during restart recovery, the entire transaction must be rolled back. The recovery code reads log record 6. The data in record 6 indicates that record 6 is an ordinary record (not a CLR), so the recovery code rolls back update 6. Then the recovery code inspects the undo-next LSN of record 6 and finds that it points to record 4'. The data in record 4' indicates that it is a CLR, so the recovery code does not roll back update 4'. Instead, it inspects the undo-next LSN of record 4' and finds that it points to record 3. Record 3 is not a CLR, so the recovery code rolls back update 3. Updates 5 and 4 are not rolled back during recovery because they were already rolled back during ordinary forward processing. Finally the recovery code rolls back updates 2 and 1.

The arrows in the following diagram illustrate how the undo-next LSN provides a mechanism that recovery code can use to skip records whose updates have already been rolled back.



CLFS Marshalling Areas

6/25/2019 • 2 minutes to read • Edit Online

A Common Log File System (CLFS) client appends log records to a *marshalling area* in volatile memory, and CLFS periodically writes those records to stable storage. A marshalling area is a collection of log I/O buffers, each of which can hold several log records. Log I/O buffers hold records that have recently been written to a stream (but possibly not flushed to stable storage) as well as records that have recently been read from the stream.

You create a marshalling area by calling **ClfsCreateMarshallingArea**, at which time you must specify the size of the log I/O buffers that the marshalling area will use and whether those buffers will be in the paged or non-paged pool. All log I/O buffers in a marshalling area are the same size, and CLFS ensures that the size is a multiple of the sector size on the underlying stable storage medium. That is, CLFS takes your requested size and rounds it up as necessary to make your I/O buffers compatible with the stable storage medium.

CLFS allocates and frees log I/O buffers as needed, but you have the option of setting the maximum number of I/O buffers that can be allocated at one time. You also have the option of providing your own buffer allocation and deallocation functions.

To specify the maximum number of log I/O buffers that can be allocated at one time for writing log records, set the *cMaxWriteBuffers* parameter of the **ClfsCreateMarshallingArea** function. Limiting the number of buffers affects the frequency of flushes to stable storage; with fewer buffers, log records must be written to stable storage more often. If you do not need to control the flush frequency, set *cMaxWriteBuffers* to INFINITE (defined in Winbase.h).

To specify the maximum number of log I/O buffers that can be allocated at one time for reading log records, set the *cMaxReadBuffers* parameter of the **ClfsCreateMarshallingArea** function. If you do not need to control the number of allocated read buffers, set *cMaxReadBuffers* to INFINITE.

If you want to do your own memory allocation for log I/O buffers, set the *pfnAllocBuffer* and *pfnFreeBuffer* parameters of the **ClfsCreateMarshallingArea** function to point to your own allocation and deallocation functions. Then CLFS will call your functions to perform the actual memory allocation and deallocation whenever it needs to create or free log I/O buffers.

In some cases, you might want to reserve space in a marshalling area ahead of time. For example, you might know that you are about to write a set of ten log records, and you want to be sure that there is enough space in the marshalling area for the entire set. To reserve space for the ten records, create a ten-element array that holds the sizes of the records, and then pass the array to the **ClfsReserveAndAppendLog** function in the *rgcbReservation* parameter. **ClfsReserveAndAppendLog** is a multi-purpose function that reserves space in a marshalling area or appends log records to a stream or does both of those things atomically. By setting the parameters appropriately, you can call **ClfsReserveAndAppendLog** to reserve space for future use without actually appending any records to the stream.

Writing Data Records to a CLFS Stream

6/25/2019 • 4 minutes to read • Edit Online

There are two types of records in a Common Log File System (CLFS) stream: data records and restart records. This topic explains how to write data records to a stream. For information about how to write restart records, see Writing Restart Records to a CLFS Stream.

Before you can write data records to a CLFS stream, you must create a marshalling area by calling **ClfsCreateMarshallingArea**. Then you can append records to the marshalling area (which is in volatile memory), and CLFS will periodically flush the records to stable storage.

There are several variations on writing data records to a stream. For example, you can reserve space ahead of time and then write several records, or you can write records without reserving space. You can request that records you write to the marshalling area be immediately queued to stable storage, or you can let CLFS queue the records according to its policy.

For all variations on writing data records, complete the following steps.

1. Create an array of one or more **CLFS_WRITE_ENTRY** structures. Each write entry structure points to a buffer that you have filled with record data.
2. Call **ClfsReserveAndAppendLog** or **ClfsReserveAndAppendLogAligned**.

The tables in the following subsections show how to set the parameters of **ClfsReserveAndAppendLog** for several variations on writing a record to a stream.

Writing a single data buffer to a stream

Suppose you have a single data buffer that you want to write to a marshalling area. You are willing to let the record be flushed to stable storage according to CLFS policy, and you do not want the record to be part of any chain of records. The following table shows how to set the parameters when you call **ClfsReserveAndAppendLog**.

PARAMETER NAME	VALUE
pvMarshalContext	A pointer to a marshalling area.
rgWriteEntries	A pointer to a **CLFS_WRITE_ENTRY** structure.
cWriteEntries	1
plsnUndoNext	CLFS_LSN_INVALID
plsnPrevious	CLFS_LSN_INVALID
cReserveRecords	0
rgcbReservation	NULL

PARAMETER NAME	VALUE
fFlags	0
plsn	A pointer to a **CLFS_LSN** structure. (This is an output parameter that receives the LSN of the record that is written.)

Reserving space for a set of CLFS log records

You can use **ClfsReserveAndAppendLog** to reserve space in a marshalling area for a set of log records without actually writing any of the records. The following table shows how to set the parameters to reserve record space.

PARAMETER NAME	VALUE
pvMarshalContext	A pointer to a marshalling area.
rgWriteEntries	NULL
cWriteEntries	0
plsnUndoNext	CLFS_LSN_INVALID
plsnPrevious	CLFS_LSN_INVALID
cReserveRecords	The number of elements in the array pointed to by *rgcbReservation*.
rgcbReservation	A pointer to an array of LONGLONG-typed variables. Each element in the array is the size, in bytes, of a record for which space will be reserved. Note that this is this size of the data portion of the record; you do not have to include the size of headers or padding.
fFlags	0
plsn	NULL

Note Another way to reserve space in a marshalling area is to call **ClfsAlignReservedLog** followed by **ClfsAllocReservedLog**.

Writing a record to reserved space

Suppose you have already reserved space for three records whose sizes, in bytes, are 100, 200, and 300. The marshalling area has a reserved-record count of 3 and enough reserved space to hold the 600 bytes of record data, the record headers, and any padding required for alignment.

Now suppose you want to write one of those records into the reserved space in the marshalling area. The available reserved space will be reduced, and the reserved-record count will be decremented from 3 to 2. The following table

shows how to set the parameters when you call **ClfsReserveAndAppendLog**.

PARAMETER NAME	VALUE
pvMarshalContext	A pointer to a marshalling area.
rgWriteEntries	A pointer to an array of **CLFS_WRITE_ENTRY** structures.
cWriteEntries	The number of elements in the array pointed to by *rgWriteEntries*.
plsnUndoNext	CLFS_LSN_INVALID or the LSN of the previous record in the undo chain. For more information about the undo chain, see CLFS Log Sequence Numbers.
plsnPrevious	CLFS_LSN_INVALID or the LSN of the previous log record in the previous-LSN chain. For more information about the previous-LSN chain, see CLFS Log Sequence Numbers.
cReserveRecords	0
rgcbReservation	NULL
fFlags	CLFS_FLAG_USE_RESERVATION
plsn	A pointer to a **CLFS_LSN** structure. (This is an output parameter that receives the LSN of the record that is written.)

Writing records with aligned entries

Suppose you want to write a record that has three write entries. The write entries vary in size between 300 and 500 bytes, but you require that each write entry begins on a 512-byte boundary. The following table shows how to set the parameters of the **ClfsReserveAndAppendLogAligned** function.

PARAMETER NAME	VALUE
pvMarshalContext	A pointer to a marshalling area.
rgWriteEntries	A pointer to an array of three CLFS_WRITE_ENTRY structures.
cWriteEntries	3
cbEntryAlignment	512

PARAMETER NAME	VALUE
plsnUndoNext	CLFS_LSN_INVALID or the LSN of the previous record in the undo chain. For more information about the undo chain, see CLFS Log Sequence Numbers.
plsnPrevious	CLFS_LSN_INVALID or the LSN of the previous log record in the previous-LSN chain. For more information about the previous-LSN chain, see CLFS Log Sequence Numbers.
cReserveRecords	0
rgcbReservation	NULL
fFlags	Zero or some combination of flags that specify flush and reservation preferences.
plsn	A pointer to a CLFS_LSN structure. (This is an output parameter that receives the LSN of the record that is written.)

The preceding tables show only a few of the many variations on reserving record space and writing records to CLFS streams. As you think of other variations, keep the following point in mind: The actions performed by **ClfsReserveAndAppendLog** (or **ClfsReserveAndAppendLogAligned**) are atomic. For example, you can make a single call to **ClfsReserveAndAppendLog** that will reserve space for a record and write the record to the stream. The pair of actions (reserve, write) will either succeed as a whole or fail as a whole.

Writing Restart Records to a CLFS Stream

6/25/2019 • 2 minutes to read • Edit Online

There are two types of records in a Common Log File System (CLFS)stream: data records and restart records. This topic explains how to write restart records to a CLFS stream. For information about how to write data records, see Writing Data Records to a CLFS Stream.

Typically, restart records are written to a stream periodically to create checkpoints that help make recovery more efficient in the event of a system failure. Assume that you have already created a marshalling area and written several data records. You can then write a restart record by calling **ClfsWriteRestartArea**. By setting the *fFlags* parameter, you can specify whether the restart record is placed in the marshalling area's reserved space or in newly allocated space.When CLFS writes a restart record to a stream, it automatically sets the previous LSN of the record to the LSN of the previously written restart record for that stream. That forms a chain of restart records that can be traversed in reverse order. For information about reading the chain of restart records, see Reading Restart Records from a CLFS Stream.

If you want to write a restart record to a stream and change the base LSN of the stream at the same time, set the *plsnBase* parameter of **ClfsWriteRestartArea** to the new base LSN.

Reading Data Records from a CLFS Stream

6/25/2019 • 7 minutes to read • Edit Online

There are two types of records in a Common Log File System (CLFS) stream: data records and restart records. This topic explains how to read a sequence of data records from a stream. For information about how to read restart records, see Reading Restart Records from a CLFS Stream.

There are several variations on reading a sequence of data records from a stream. You can read forward in the stream from a specified record or you can read backward along a chain of linked records.

For all variations on reading a sequence of data records, complete the following steps.

1. Call **ClfsReadLogRecord** to obtain a read context and the first data record in the sequence.
2. Pass the read context you obtained in step 1 to **ClfsReadNextLogRecord** repeatedly to obtain the remaining data records in the sequence.

Caution Read contexts are not thread-safe. Clients are responsible for serializing access to read contexts.

The following subtopics discuss the details of reading the different types of record sequences and chains.

Reading forward from a specified data record

To read forward in a CLSF stream (starting at the data record of your choice), you must create a read context that has its mode set to **ClfsContextForward**. To create a read context and read the first record (in the set that you have chosen to read), call **ClfsReadLogRecord** as shown in the following table.

PARAMETER NAME	VALUE
pvMarshalContext	Supply a pointer to a marshalling area.
plsnFirst	Supply the LSN of the first record you want to read. This must be the LSN of a data record, not a restart record.
peContextMode	Supply the value **ClfsContextForward**.
ppvReadBuffer	Receive your record data.
pcbReadBuffer	Receive the size of your record data.
peRecordType	Receive the record type. This value is a set of flags that indicate various features of the record. The record is a data record, so the value you receive should have the ClfsDataRecord flag set and the ClfsRestartRecord flag clear.
plsnUndoNext	Receive the undo-next LSN of the data record. You do not need this value to continue reading the chain, so you can ignore it.

PARAMETER NAME	VALUE
plsnPrevious	Receive the previous LSN of the data record. You do not need this value to continue reading the chain, so you can ignore it.
ppvReadContext	Receive a pointer to an opaque read context. Use the read context to read subsequent records.

After you have obtained the read context and the first record, you can obtain subsequent records in the stream by calling **ClfsReadNextLogRecord** repeatedly. When there are no more data records in the stream, **ClfsReadNextLogRecord** returns STATUS_END_OF_FILE. The following table shows how to set and interpret the parameters.

PARAMETER NAME	VALUE
pvReadContext	Supply a pointer to the read context you received from **ClfsReadLogRecord**.
ppvBuffer	Receive your record data.
pcbBuffer	Receive the size of your record data.
peRecordType	Supply the value of **ClfsDataRecord**.
plsnUndoNext	Receive the undo-next LSN field of the data record. You do not need this value to continue reading the chain, so you can ignore it.
plsnPrevious	Receive the previous-LSN field of the data record. You do not need this value to continue reading the chain, so you can ignore it.
plsnRecord	Receive the LSN of the data record that was read.

Reading a chain of data records linked by the previous LSN

When you write a data record to a CLFS stream, you can set the previous LSN of the data record to the LSN of any record that you previously wrote to the stream. By setting the previous LSN, you can create a chain of related records that can later be traversed in reverse order. For example, suppose you are performing a database transaction and you must write several CLFS log records to describe the updates made by the transaction. Each time you write a log record that describes a transaction update, you could set the previous LSN of the record to the LSN of the previous log record that describes an update made by the same transaction.

Suppose you have written a chain of data records that are linked by their previous LSNs. To read the chain of records, you must create a read context that has its mode set to **ClfsContextPrevious**. To create a read context and read the first record in the chain, call **ClfsReadLogRecord** as shown in the following table.

PARAMETER NAME	VALUE
pvMarshalContext	Supply a pointer to a marshalling area.
plsnFirst	Supply the LSN of the first record in the chain. This must be the LSN of a data record, not a restart record.
peContextMode	Supply the value of **ClfsContextPrevious**.
ppvReadBuffer	Receive your record data.
pcbReadBuffer	Receive the size of your record data.
peRecordType	Receive the record type. This value is a set of flags that indicate various features of the record. The record is a data record, so the value you receive should have the ClfsDataRecord flag set and the ClfsRestartRecord flag clear.
plsnUndoNext	Receive the undo-next LSN of the data record. You do not need this value to continue reading the chain, so you can ignore it.
plsnPrevious	Receive the previous LSN of the data record. You do not need this value to continue reading the chain, so you can ignore it.
ppvReadContext	Receive a pointer to an opaque read context. Use the read context to read the previous records in the chain.

After you have the read context and the first record, you can read the remaining records in the chain by calling **ClfsReadNextLogRecord** repeatedly. The following table shows how to set and interpret the parameters.

PARAMETER NAME	VALUE
pvReadContext	Supply a pointer to the read context you received from **ClfsReadLogRecord**.
ppvBuffer	Receive your record data.
pcbBuffer	Receive the size of your record data.
peRecordType	Supply the value of **ClfsDataRecord**.

PARAMETER NAME	VALUE
plsnUndoNext	Receive the undo-next LSN of the data record. You do not need this value to continue reading the chain, so you can ignore it.
plsnPrevious	Receive the previous LSN of the data record. You do not need this value to continue reading the chain, so you can ignore it.
plsnRecord	Receive the LSN of the data record that was read.

As you make repeated calls to **ClfsReadNextLogRecord**, your sequence of calls will end in one of the following ways.

- Eventually you will read a data record that has its previous LSN set to CLFS_LSN_INVALID. The next time you call **ClfsReadNextLogRecord**, it will return STATUS_END_OF_FILE.
- Eventually you will read a data record that has a previous LSN that is less than both the base LSN of the stream and the *archive tail* of the stream. The next time you call **ClfsReadNextLogRecord**, it will return STATUS_LOG_START_OF_LOG.

Reading a chain of data records linked by the undo-next LSN

When you write a data record to a CLFS stream, you can set the undo-next LSN of the data record to the LSN of any record that you previously wrote to the stream. By setting the undo-next LSN, you can create a chain of related records that can be traversed in reverse order. For more information about creating and interpreting undo-next chains, see CLFS Log Sequence Numbers.

Suppose you have written a chain of data records that are linked by their undo-next LSNs. To read the chain of records, you must call **ClfsReadLogRecord** to create a read context that has its mode set to **ClfsContextUndoNext**. After that, the process is identical to reading a chain linked by previous LSNs (described previously in this topic).

Reading a chain of data records linked by the user LSN

In addition to chains linked by previous LSNs and undo-next LSNs, you can create chains linked by your own LSNs that you embed in your record data.

Suppose you have written a chain of data records that are linked by LSNs you have stored in the record data itself. To read the chain of records, you must create a read context that has its mode set to either **ClfsContextPrevious** or **ClfsContextUndoNext**. Create your read context and obtain the most recently written record in the chain by calling **ClfsReadLogRecord**. Then call **ClfsReadNextLogRecord** repeatedly to obtain the previous records in the chain. Each time you call **ClfsReadNextLogRecord**, set the *plsnUser* parameter to the LSN of the previous record in your chain. The LSN you supply in *plsnUser* overrides any values stored in the current record's previous-LSN or undo-next LSN fields.

Note that you can only move backward in the stream when you call **ClfsReadNextLogRecord** to read a record chain. The LSN you supply in *plsnUser* must be less than the LSN of the current record in the chain.

Reading Restart Records from a CLFS Stream

6/25/2019 • 2 minutes to read • Edit Online

To read all of the restart records in a Common Log File System (CLFS) stream (in reverse order), use the following procedure.

1. Call **ClfsReadRestartArea** to obtain a read context and the restart record that was most recently written to the stream.
2. Pass the read context you obtained in step 1 to **ClfsReadPreviousRestartArea** repeatedly to obtain the remaining restart records in the log.

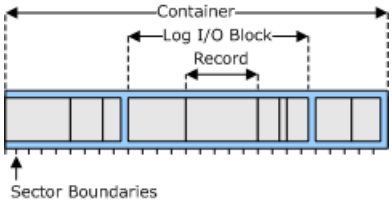
Note When you call **ClfsWriteRestartArea** to write a restart record to a stream, CLFS automatically sets the previous LSN of that record to the LSN of the previous restart record in the stream. Those previous LSNs form the chain that is followed by repeated calls to **ClfsReadPreviousRestartArea**.

CLFS Stable Storage

6/25/2019 • 3 minutes to read • Edit Online

When you write a record to a Common Log File System (CLFS) stream, the record is placed in a log I/O block (in a marshalling area) in volatile memory. Periodically, CLFS flushes log I/O blocks from the marshalling area to stable storage such as a disk. On the stable storage device, the log consists of a set of containers, each of which is a contiguous extent on the physical medium. A collection of containers that forms the stable storage for a stream is called a *log*, or a *physical log*.

The following figure illustrates a container.



The preceding figure illustrates a container that holds three log I/O blocks. The first log I/O block contains three records, the second contains five records, and the third contains two records. As the figure suggests, the beginning of each log I/O block is always aligned with the beginning of a sector on the stable storage medium. Note that log I/O blocks on stable storage vary in size.

CLFS uses a set of three numbers to locate a record in a log.

- The *container identifier* identifies the container that holds the record.
- The *block offset* gives the byte offset, within the container, of the beginning of the log I/O block that holds the record.
- The *record sequence* number identifies the record within the log I/O block.

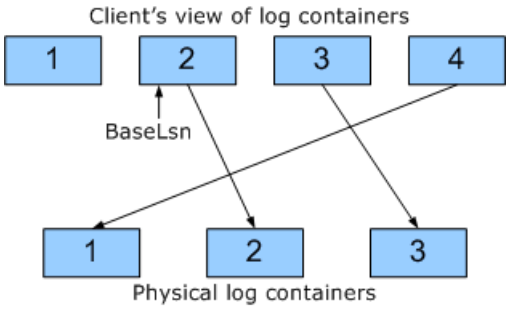
The log sequence number (LSN) of a CLFS log record actually holds those three pieces of information: container identifier, block offset, and record sequence number. However, the LSNs given to log clients contain *logical container identifiers* that CLFS must map to physical container identifiers before it accesses the records on stable storage.

CLFS uses logical container identifiers to give clients the view that log records are being written to an ongoing sequence of containers, when in fact, the physical containers are being recycled.

Suppose a log has three containers, and a single client is writing CLFS records to the log. The following scenario shows how a container could be recycled.

1. The client writes enough log records to fill all three containers.
2. The client sets the log base (by calling **ClfsAdvanceLogBase** or **ClfsWriteRestartArea**.) to one of the records in container 2. By doing that, the client is saying that it no longer needs the records in container 1.
3. The client writes another record to the log and gets back the LSN of the newly written record. The logical container identifier in that LSN is 4. When records are flushed to stable storage, records that the client sees in logical container 4 will go to physical container 1.

The following figure illustrates the scenario; it shows how the client sequence of logical containers is mapped to physical containers on stable storage.



The logical container identifier, block offset, and record sequence number are stored in an LSN in such a way that the LSNs for a particular stream always form a strictly increasing sequence. That is, the LSN (with logical container identifier) of a log record written to a stream is always greater than the LSNs of the log records previously written to that same stream. LSNs, then, serve a dual purpose: 1) they give the clients of a stream an ordered sequence of record identifiers, and 2) they provide CLFS with the location of records on stable storage.

Given the LSN of a record, you can extract the logical container identifier, the block offset, and the record sequence number by calling the following functions.

ClfsLsnContainer

ClfsLsnBlockOffset

ClfsLsnRecordSequence

The logical container identifier is a 32-bit number, so there are 2^32 possible logical container identifiers, and they are in the range 0x0 through 0xFFFFFFFF. A stream can have at most 2^32 logical containers.

The block offset is stored in 23 bits of the LSN, but **ClfsLsnBlockOffset** returns a 32-bit number that is aligned with the sector size of the stable storage medium. The block offset is always a multiple of 512. Also, the block offset is aligned with the sector size of the stable storage medium. For example, if the sector size is 1024 bytes, the block offset will be a multiple of 1024.

The record sequence number is a 9-bit number, so there are 2^9 (512) possible record sequence numbers, and they are in the range 0x0 through 0x1FF. A log I/O block can have at most 512 records.

Dedicated CLFS Logs

6/25/2019 • 2 minutes to read • Edit Online

A Common Log File System (CLFS) log can be either dedicated or multiplexed. A *dedicated log* serves as stable storage for a single stream. A *multiplexed log* serves as stable storage for several streams. This topic discusses dedicated logs. For information about multiplexed logs, see Multiplexed CLFS Logs.

To create a dedicated log, perform the following steps.

1. Call **ClfsCreateLogFile** to obtain a pointer to a **LOG_FILE_OBJECT** structure. Set the *puszLogFileName* parameter to a string of the form "log:<*log name*>" where <*log name*> is a valid path on the underlying file system. For example, if you set *puszLogFileName* to "log:c:\ClfsLogs\myLog", the base log file myLog.blf would be created in the c:\ClfsLogs directory. The c:\ClfsLogs directory would also serve as the default location for containers that you add to the log later.

   Note It is the form of the string passed in *puszLogFileName* that determines whether CLFS creates a dedicated or multiplexed log. If the string has a double colon (::) after the log name, then CLFS creates a multiplexed log. In the example given here, "log:c\ClfsLogs\myLog" has no double colon, so CLFS creates a dedicated log.

The **LOG_FILE_OBJECT** pointer returned by **ClfsCreateLogFile** represents an open instance of the dedicated log's one and only stream.

2. Pass the **LOG_FILE_OBJECT** pointer you obtained from **ClfsCreateLogFile** to **ClfsAddLogContainer** to create a container (contiguous physical extent) on stable storage that will hold log records. Specify the size of the container (which will be rounded up to a multiple of 512 kilobytes) by setting the *pcbContainer* parameter. Set the *puszContainerPath* parameter to specify a path name for the container. The path name can be absolute or relative to the directory that contains the base log file.

   You can create additional containers for your log by calling **ClfsAddLogContainer** again. Note that all containers for a given log must be the same size. As an alternative to calling **ClfsAddLogContainer** several times, you can call **ClfsAddLogContainerSet** to create several containers simultaneously.

3. Pass the **LOG_FILE_OBJECT** pointer you obtained from **ClfsCreateLogFile** to **ClfsCreateMarshallingArea** to obtain a pointer to a marshalling area that you can use to read and write log records to your stream. Specify the size of the log I/O blocks that the marshalling area will use by setting the *cbMarshallingBuffer* parameter. There are several other parameters you can use to set various properties of the marshalling area.

   If you need additional marshalling areas, pass the same **LOG_FILE_OBJECT** pointer to **ClfsCreateMarshallingArea** again, once for each additional marshalling area that you need.

Now that you have one or more marshalling areas associated with your stream, you can write records to those marshalling areas by calling the following functions.

ClfsReserveAndAppendLog

ClfsReserveAndAppendLogAligned

ClfsWriteRestartArea

Each time you write a record, you get back a log sequence number (LSN) that identifies the record. The LSN assigned to a record is always greater than the LSN assigned to the previously written record, regardless of which marshalling area was used to write the record.

Multiplexed CLFS Logs

6/25/2019 • 4 minutes to read • Edit Online

A *multiplexed log* serves as stable storage for several streams. A *dedicated log* serves as stable storage for a single stream. This topic discusses multiplexed logs. For information about dedicated logs, see Dedicated CLFS Logs.

Each stream of a multiplexed log provides its clients with the illusion that their stream is the entire log. A client, in this context, is a driver, a thread, or some other unit of software that writes to and reads from a Common Log File System (CLFS) log. It is possible for a single stream to have several clients. Each client would have its own **LOG_FILE_OBJECT** structure, which represents an open instance of the stream.

Consider the case of a multiplexed log that has two streams, each of which has one client. You can use the following procedure to create the log, the streams, and the client marshalling areas.

1. On behalf of client 1, call **ClfsCreateLogFile** to obtain a pointer to a **LOG_FILE_OBJECT** structure. Set the *puszLogFileName* parameter to a string of the form "log:<log name>::<stream name>" where <log name> is a valid path on the underlying file system, and <stream name> is the name that you have chosen to give to the stream that will be used by client 1. For example, you could set *puszLogFileName* to "log:c:\ClfsLogs\myLog::Stream1". In that case, CLFS would create the base log file myLog.blf in the c:\ClfsLogs directory, and Stream1 would be the name of the stream used by client 1.

   Note It is the form of the string passed in *puszLogFileName* that determines whether CLFS creates a dedicated or multiplexed log. If the string has a double colon (::) after the path name, then CLFS creates a multiplexed log.

2. On behalf of client 2, call **ClfsCreateLogFile** to obtain a pointer to a **LOG_FILE_OBJECT** structure. Set the *puszLogFileName* parameter to a string of the form "log:<log name>::<stream name>" where <log name> is the same path name you used for client 1, and <stream name> is the name that you have chosen to give to the stream that will be used by client 2. For example, you could set *puszLogFileName* to "log:c:\ClfsLogs\myLog::Stream2".

3. Pass one of the **LOG_FILE_OBJECT** pointers you obtained from **ClfsCreateLogFile** to **ClfsAddLogContainer** to create a container (contiguous physical extent) on stable storage that will hold log records. Specify the size of the container (which will be rounded up to a multiple of 1 megabyte) by setting the *pcbContainer* parameter. Set the *puszContainerPath* parameter to specify a path name for the container. The path name can be absolute or relative to the directory that contains the base log file.

   You can create additional containers for your log by calling **ClfsAddLogContainer** again. Note that all containers for a given log must be the same size. As an alternative to calling **ClfsAddLogContainer** several times, you can call **ClfsAddLogContainerSet** to create several containers simultaneously. Note that your set of containers will serve as stable storage for log records written by both client 1 and client 2.

4. Pass the **LOG_FILE_OBJECT** pointer you obtained on behalf of client 1 to **ClfsCreateMarshallingArea** to obtain a pointer to a marshalling area that client 1 can use to read and write log records. Specify the size of the log I/O blocks that the marshalling area will use by setting the *cbMarshallingBuffer* parameter. There are several other parameters you can use to set various properties of the marshalling area.

   If client 1 needs additional marshalling areas, pass the same **LOG_FILE_OBJECT** pointer to **ClfsCreateMarshallingArea** again, once for each additional marshalling area that client 1 needs.

5. Pass the **LOG_FILE_OBJECT** pointer you obtained on behalf of client 2 to **ClfsCreateMarshallingArea** to obtain a marshalling area that client 2 can use to read and write log records. Specify the size of the log I/O blocks that the marshalling area will use by setting the *cbMarshallingBuffer* parameter.

Note There are several other parameters you can use to set various properties of the marshalling area.

If client 2 needs additional marshalling areas, pass the same **LOG_FILE_OBJECT** pointer to **ClfsCreateMarshallingArea** again, once for each additional marshalling area that client 2 needs.

Now that clients 1 and 2 each have a **LOG_FILE_OBJECT** and one or more marshalling areas, they can each write records to their own streams (by way of the marshalling areas associated with those streams) by calling the following functions.

ClfsReserveAndAppendLog

ClfsReserveAndAppendLogAligned

ClfsWriteRestartArea

All log records written by clients 1 and 2 go to the same log; that is, to the same set of physical containers on stable storage. CLFS multiplexes the log records written by the two clients and keeps track of which records belong to each stream.

As client 1 writes records to its stream, it gets back an increasing sequence of log sequence numbers (LSNs) that identify those records. Similarly, client 2 gets its own sequence of LSNs. The LSNs that belong to a particular stream can be compared to determine the order in which the corresponding records were written. However, an LSN that belongs to one stream cannot be compared to an LSN that belongs to another stream.

CLFS maintains a base LSN and a last LSN for every stream, including streams that share a multiplexed log. Each stream has an active portion that begins with the record pointed to by the base LSN and ends with the record pointed to by the last LSN. Note that a container in a multiplexed log on stable storage cannot be recycled until the base LSNs of all the log's streams have advanced beyond any records stored in that container.

CLFS Support for Archiving

6/25/2019 • 2 minutes to read • Edit Online

Common Log File System (CLFS) supports archiving for dedicated logs by maintaining an archive tail. When you call **ClfsCreateLogFile** to create a dedicated log, you can set the FILE_ATTRIBUTE_ARCHIVE flag of the *fFlagsAndAttributes* parameter to specify that CLFS should maintain an archive tail for the log. A log for which CLFS maintains an archive tail is called a *non-ephemeral log*.

Suppose you are performing transactions on a database and each transaction has several updates that are described by log records. After a particular transaction has committed and been written to stable storage, you might not need the log records that describe that transaction any more. That is, the log records would not be needed during restart recovery in the event of a system failure. However, if the stable storage medium that holds the database fails and the database has not been recently archived on a different medium, the database updates could be lost.

The preceding paragraph describes archiving database records, but in other scenarios you might want to archive log records. In either case, archiving is the responsibility of the clients (your software). You can keep track of the archiving you have done by setting the log's archive tail. The archive tail is the log sequence number (LSN) of the oldest record for which archiving has not yet been completed.

A non-ephemeral log actually has two tails: one marked by the base LSN and one marked by the archive tail. You can position the two tails as you see fit by calling **ClfsAdvanceLogBase** (or **ClfsWriteRestartArea**), and **ClfsSetArchiveTail**. Typically the base LSN points to the oldest record that would still be needed for transaction rollback or restart recovery, and the archive tail points to the oldest record for which archiving has not been performed. Note that the archive tail might be less than the base LSN or it might be greater than the base LSN.

The base LSN and the archive tail are important when you call **ClfsReadNextLogRecord** repeatedly to read a chain of records linked by previous LSNs, undo-next LSNs, or user LSNs. **ClfsReadNextLogRecord** will not read a record whose LSN is less than both the archive tail and the base LSN. It will, however, read a record whose LSN is between the archive tail and the base LSN. For more information about following record chains, see Reading Data Records from a CLFS Stream.

Introduction to KTM

12/5/2018 • 2 minutes to read • Edit Online

The Kernel transaction manager (KTM) is a transaction management service that enables you to create a transaction processing system (TPS) in user mode or kernel mode (or both).

KTM is a kernel-mode service of the Microsoft Windows operating system. KTM is available on Windows Vista and later versions of Windows.

KTM provides both user-mode and kernel-mode interfaces. This documentation describes the kernel-mode interfaces of KTM. For information about the Microsoft Win32 user-mode interfaces, see the Microsoft Windows SDK.

For more information about transaction theory, see the book titled *Transaction Processing: Concepts and Techniques* by Jim Gray and Andreas Reuter.

This section includes the following topics:

When to Use Kernel-Mode KTM

12/5/2018 • 2 minutes to read • Edit Online

You can use kernel-mode KTM with your kernel-mode component to support transacted operations in kernel mode, or to coordinate transacted operations between a kernel-mode component that uses kernel-mode KTM and a user-mode component that uses user-mode KTM.

For example, you might use KTM in the following situations:

- Your kernel-mode driver must open a file, modify the file's contents, and save the modified file, and it must prevent damage to the file if a write operation fails. If your driver performs these operations within a transaction, the driver does not have to copy and rename the old file, modify the new copy, delete the old file, and then rename the new copy.
- You are designing a new data storage system that stores information in one or more databases. Components of your system will access the databases in kernel mode, or possibly in both user mode and kernel mode. Transactional clients of your system will encapsulate their database operations within transactions so that all modifications to all databases either succeed or fail as a unit.

Transaction Processing Terms

12/5/2018 • 4 minutes to read • Edit Online

Before you begin to use KTM, you should know the definitions of the following terms: transaction, resource manager, transactional client, transaction manager, log stream, enlistment, and transaction processing system.

transaction
A *transaction* is a collection of data operations. All the operations must succeed for the transaction to succeed. If all the operations succeed, the transaction can be *committed* (that is, its results can be made permanent and public). If any operation fails, the transaction must be *rolled back*, (that is, all changes must be removed so that the data is in the same state that it was in before the transaction's operations began).

A transaction's operations are *atomic*, *consistent*, *isolated*, and *durable* (ACID).

- They are atomic because they must be committed or rolled back as a whole.
- They are consistent because the operations always produce an accurate result, whether they are committed or rolled back.
- They are isolated because each transaction's results are not visible to other transactions until the transaction's operations have been committed or rolled back.
- They are durable because, after the transaction's operations have been committed or rolled back, the results of the operations are permanent.

An example of a transaction is the set of operations that must be performed when you use an automatic teller machine (ATM) to transfer money from your checking account to your savings account. The debit from your checking account and the credit to your savings account must appear to be a single, atomic operation.

An operation that is part of a transaction is also known as a *transacted operation*.

resource manager
A *resource manager* is a software component that manages data resources that can be updated by transacted operations. For example, if you are designing a database system, you might provide a resource manager that stores and retrieves the database's data. A simple transaction processing system (TPS) might have only one resource manager.

A resource manager typically also provides a public interface that transactional clients can call to access the resource manager's data. For example, the resource manager for a database might provide a set of functions that clients can call to read from and write to the database.

A more complex TPS can have multiple resource managers, each of which manages a separate database or other resource while participating in the system's transactions.

For more information about resource managers, see Creating a Resource Manager.

In some cases, one resource manager is *superior* to the other resource managers and can initiate commit operations. In KTM, such resource managers are called superior transaction managers.

transactional client
A *transactional client* is a software component that accesses a database that a resource manager supports, typically by calling functions that the resource manager exports. The client is responsible for creating transactions, performing a set of operations that a resource manager supports, and then informing the transaction manager (KTM) that the transaction should be either committed or rolled back.

For more information about transactional clients, see Creating a Transactional Client.

transaction manager

A *transaction manager*, such as KTM, provides the infrastructure that enables transactional clients and resource managers to communicate with each other. It also tracks the state of each transaction (but not the data that clients and resource managers handle).

The transaction manager can also coordinate recovery operations after a system crash.

The transaction manager has no knowledge of the data or operations that make up a transaction. The data and operations are controlled by the clients and resource managers.

KTM provides functions that transactional clients can call. These functions enable clients to create, commit, and roll back transactions.

KTM also provides functions that resource managers can call. These functions enable resource managers to enlist in transactions so that they can receive notifications about transactions. After a resource manager enlists in a transaction, it can receive a notification when a transactional client is ready to commit or roll back the transaction, or when a recovery operation occurs.

log stream

A *log stream* is a recorded history of the events that have happened to transactions. KTM maintains a log stream by using the Common Log File System (CLFS). KTM records state changes for each transaction so that it can support rollback and recovery operations when they are necessary.

Resource managers must also use a log stream to record data and operations.

A rollback operation requires KTM and resource managers to restore a transaction and all data to an initial state. KTM and resource managers record the initial state of each transaction in the log streams so that they can fetch it during a rollback operation.

Recovery operations occur after a system crash. When the operating system subsequently restarts, KTM and resource managers can use log stream contents to rebuild a transaction's state to the state that it was in before the crash.

For more information about log streams in KTM, see Using Log Streams with KTM.

enlistment

An *enlistment* is an association between a resource manager and a transaction. KTM provides a set of functions that resource managers call to create and manage enlistments. After a resource manager creates an enlistment, KTM sends notifications to the resource manager when the transaction's state changes.

transaction processing system

A *transaction processing system* (TPS) is a collection of a transaction manager, one or more resource managers, one or more log streams, and one or more transactional clients that access the resource managers' resources.

Understanding TPS Components

6/25/2019 • 7 minutes to read • Edit Online

Any *transaction processing system* (TPS) that uses the Kernel Transaction Manager (KTM) and the Common Log File System (CLFS) should contain the following important components:

- A *transaction manager* (KTM)

  KTM tracks the state of each transaction and coordinates recovery operations after a system crash.

- One or more *resource managers*

  Resource managers, which you provide, manage the data that is associated with each transaction.

- One or more CLFS *log streams*

  The transaction manager and resource managers use CLFS log streams to record information that can be used to commit, roll back, or recover a transaction.

- One or more *transactional clients*

  Typically, each transactional client of your TPS can create a transaction, perform operations on data within the context of the transaction, and then initiate either a commit or rollback operation for the transaction.

This topic introduces you to a simple TPS with one resource manager, a more complex TPS that contains multiple resource managers, and some other TPS scenarios.

The Using KTM section provides detailed information about how to use KTM to create TPS components.

Simple TPS

A simple TPS might consist of KTM, one resource manager, and CLFS. Transactional clients can communicate with the resource manager by an interface that the resource manager provides.

For example, suppose that you want to create a database management system. You want your system's clients to access the database by opening a handle to a database object, performing read and write operations on the object, and then closing the object handle.

Now suppose that you want sets of read and write operations to occur atomically so that other users of the system see only the final result. You can achieve that goal by designing a TPS that enables clients to bind sets of database operations to a transaction.

Your system should include a resource manager that manages the data in the database in response to read and write requests from clients. This resource manager could export an application programming interface (API) that enables clients to associate a transaction with a set of read and write operations.

When your resource manager is loaded, it must register itself with KTM by calling **ZwCreateTransactionManager** and **ZwCreateResourceManager**. Then, the resource manager can participate in transactions.

You might want your resource manager to support a set of functions that enable clients to create data objects, read and write data that is associated with the data objects, and close the data objects. The following pseudocode shows an example code sequence from a client.

```
CreateDataObject (IN TransactionID, OUT DataHandle);
ReadData (IN DataHandle, OUT Data);
WriteData (IN DataHandle, IN Data);
WriteData (IN DataHandle, IN Data);
WriteData (IN DataHandle, IN Data);
CloseDataObject (IN DataHandle);
```

Before a client can call your resource manager's *CreateDataObject* routine, the client must create a transaction object by calling KTM's **ZwCreateTransaction** routine and obtain the transaction object's identifier by calling **ZwQueryInformationTransaction**.

When the client calls your resource manager's *CreateDataObject* routine, the client passes the transaction object's identifier to the resource manager. The resource manager can call **ZwOpenTransaction** to obtain a handle to the transaction object, and then it can call **ZwCreateEnlistment** to register its participation in the transaction.

At this point, the client can start performing operations on the data object. Because the client provided a transaction identifier when it created the data object, the resource manager can assign all the read and write operations to the transaction.

Your resource manager must record all the results of data operations that the client specifies without making the results permanent. Typically, the resource manager uses CLFS to record the operation results in a transaction log stream.

When the client has finished calling the resource manager to perform transactional operations, it calls KTM's **ZwCommitTransaction** routine. At this point, KTM notifies the resource manager that it should make the operations permanent. The resource manager then moves the operation results from the log stream to the data's permanent storage medium. Finally, the resource manager calls **ZwCommitComplete** to inform KTM that the commit operation is complete.

What happens if your resource manager reports an error for one of the client's calls to *ReadData* or *WriteData*? The client can call **ZwRollbackTransaction** to roll back the transaction. As a result of that call, KTM notifies the resource manager that it should restore the data to its original state. Then, the client can either create a new transaction for the same operations, or it can choose to not continue.

The following pseudocode shows an example of a more detailed sequence of a client's transactional operations.

```
    ZwCreateTransaction (&TransactionHandle, ...);
    ZwQueryInformationTransaction (TransactionHandle, ...);
    CreateDataObject (TransactionID, &DataHandle);
    Status = ReadData (DataHandle, &Data1);
    if (Status == Error) goto ErrorRollback;
    Status = WriteData (DataHandle, Data2);
    if (Status == Error) goto ErrorRollback;
    Status = WriteData (DataHandle, Data3);
    if (Status == Error) goto ErrorRollback;
    Status = WriteData (DataHandle, Data4);
    if (Status == Error) goto ErrorRollback;
    ZwCommitTransaction (TransactionHandle, ...);
    goto Leave;
ErrorRollback:
    ZwRollbackTransaction (TransactionHandle, ...);
Leave:
    ZwClose (TransactionHandle);
    return;
```

What happens if the system crashes after the transaction is created but before it is committed or rolled back? Every time that your resource manager loads, it should call **ZwRecoverTransactionManager** and **ZwRecoverResourceManager**. Calling **ZwRecoverTransactionManager** causes KTM to open its log stream and read the transaction history. Calling **ZwRecoverResourceManager** causes KTM to notify the resource

manager of any enlisted transactions that were in progress before the crash and which transactions the resource manager must therefore recover.

If a transactional client called **ZwCommitTransaction** for a transaction before the crash, and began to handle commit operations for the transaction, the resource manager must be able to restore the transaction's state to the point immediately prior to the crash. If the client was not ready to commit the transaction before the crash, the resource manager can discard the data and roll back the transaction.

For more information about how to write transactional clients, see Creating a Transactional Client.

For more information about how to write resource managers, see Creating a Resource Manager.

Multiple Resource Managers in a TPS

Now suppose that your TPS enables clients to modify information in two separate databases within a single transaction, so that the transaction succeeds only if the modifications of both databases succeed.

In this case, your TPS can have two resource managers, one for each database. Each resource manager can export an API that clients can use to access the resource manager's database.

The following pseudocode shows how a client might create a single transaction that contains operations on two databases that two resource managers support.

In this example, the client reads data from the first database and writes it to the second database. Then, the client reads data from the second database and writes it to the first database. (The first resource manager exports functions that begin with **Rm1**, and the second resource manager exports functions that begin with **Rm2**.)

```
    ZwCreateTransaction (&TransactionHandle, ...);
    ZwQueryInformationTransaction (TransactionHandle, ...);
    Rm1CreateDataObject (TransactionID, &Rm1DataHandle);
    Rm2CreateDataObject (TransactionID, &Rm2DataHandle);
    Status = Rm1ReadData (Rm1DataHandle, &Rm1Data);
    if (Status == Error) goto ErrorRollback;
    Status = Rm2WriteData (Rm2DataHandle, Rm1Data);
    if (Status == Error) goto ErrorRollback;
    Status = Rm2ReadData (Rm2DataHandle, &Rm2Data);
    if (Status == Error) goto ErrorRollback;
    Status = Rm1WriteData (Rm1DataHandle, Rm2Data);
    if (Status == Error) goto ErrorRollback;
    ZwCommitTransaction (TransactionHandle, ...);
    goto Leave;
ErrorRollback:
    ZwRollbackTransaction (TransactionHandle, ...);
Leave:
    ZwClose (TransactionHandle);
    return;
```

Because the client passes the same transaction identifier to both resource managers, both resource managers can call **ZwOpenTransaction** and **ZwCreateEnlistment** to enlist in the transaction. When the client eventually calls **ZwCommitTransaction**, KTM notifies each resource manager that the manager should make the operations permanent, and each resource manager calls **ZwCommitComplete** when it has finished.

Other TPS Scenarios

KTM supports other TPS scenarios. For example, the following scenarios describe components that a TPS might contain:

- One resource manager that manages multiple databases.

  The resource manager's API could enable clients to open and access more than one database at a time, and the client could combine accesses to multiple databases in a single transaction.

- One resource manager with an API that clients call, and additional resource managers with APIs that the first resource manager calls.

  The client communicates only with the first resource manager. When that resource manager processes requests from a client, it can access the additional resource managers, as needed, to process the client's requests. For example, a resource manager manages a client-accessible database that requires backup or data verification operations from a second resource manager that is not available to clients.

- An existing client and resource manager that do not use KTM, integrated with an additional set of resource managers that do use KTM.

  In this case, you typically have to modify the existing resource manager so that it becomes a superior transaction manager that communicates with KTM.

Additional Transactional Interfaces

6/25/2019 • 2 minutes to read • Edit Online

In addition to the transactional interfaces that you can use by accessing KTM, Microsoft provides several additional transactional interfaces, including the following:

- For file system minifilter drivers, the filter manager provides routines that enable minifilter drivers to enlist in transactions, receive notification about transaction state changes, and attach contexts to transactions. For more information about these capabilities, see **FltEnlistInTransaction**.
- Beginning with Windows Vista, the NTFS file system and the registry are implemented as resource managers that support transactional operations. For more information about transactional NTFS and transactional registry capabilities, see the Microsoft Windows SDK.
- The Distributed Transaction Coordinator (DTC) provides interoperability with KTM through the **IKernelTransaction** interface. For more information about the **IKernelTransaction** interface, see the Microsoft Windows SDK.
- The .NET Framework supports the **System.Transactions** namespace. For more information about this namespace, see the Microsoft developer website.

KTM Objects

6/25/2019 • 2 minutes to read • Edit Online

The Kernel Transaction Manager (KTM) defines the following four object types:

- Transaction manager objects, which KTM uses to maintain memory-resident information about a *log stream* for a *transaction processing system* (TPS).
- Resource manager objects, which represent the *resource managers* within a TPS.
- Transaction objects, which represent the transactions that *transactional clients* create.
- Enlistment objects, which represent *enlistments* that provide connections between transactions and resource managers.

These four object types all have the following characteristics:

- To create an object and obtain an object handle, TPS components can call a *create* routine.
- To obtain additional object handles to an existing object, TPS components can call an *open* routine.
- To obtain information about an object, TPS components can call a *query* routine.
- To close an object handle, TPS components call **ZwClose**.

KTM assigns an identifier GUID to each object. For transaction objects, this identifier GUID is also known as a *unit of work (UOW) identifier* that clients can specify. TPS components can use the identifier GUIDs to track objects. A TPS component that creates an object can pass the object's identifier GUID to another component so that the latter component can open a handle to the object.

Any TPS component that uses KTM can call **ZwEnumerateTransactionObject** to enumerate KTM objects, but most components do not have to call this routine.

This section contains the following topics:

Transaction Manager Objects

6/25/2019 • 2 minutes to read • Edit Online

The main purpose of the *transaction manager object* is to create and maintain a Common Log File System (CLFS) log stream that KTM uses to record status information about transactions.

The transaction manager object also contains a virtual clock value that KTM maintains and uses to sequence information in the object's log stream.

KTM provides a set of transaction manager object routines that kernel-mode TPS components can call. KTM also provides a similar set of user-mode routines that user-mode applications can call. For more information about the user-mode routines, see the Microsoft Windows SDK.

KTM creates a transaction manager object when a resource manager calls **ZwCreateTransactionManager**. Typically, each resource manager in a TPS creates a transaction manager object. But you can also design a TPS in which several resource managers share a single transaction manager object.

TPS components can open additional handles to an existing transaction manager object by calling **ZwOpenTransactionManager**. For example, if your TPS has several resource managers that share a single transaction manager object, one resource manager calls **ZwCreateTransactionManager** and then passes the object GUID to the other resource managers so that they can call **ZwOpenTransactionManager**.

Resource managers close their handles to transaction manager objects by calling **ZwClose**.

The operating system deletes the object after the last handle is closed and KTM has released all its references to the object.

Resource Manager Objects

6/25/2019 • 2 minutes to read • Edit Online

Resource manager objects represent resource managers. Each resource manager must call **ZwCreateResourceManager** to register itself to KTM.

KTM provides a set of resource manager object routines that kernel-mode resource managers can call. KTM also provides a similar set of user-mode routines that user-mode applications can call. For more information about the user-mode routines, see the Microsoft Windows SDK.

KTM creates a resource manager object when a resource manager calls **ZwCreateResourceManager**.

TPS components can call **ZwOpenResourceManager** to open additional handles to a resource manager object. But most TPS designs do not require additional open handles.

Resource managers close their handles to resource manager objects by calling **ZwClose**. If the last handle is closed, and if the resource manager still has enlistments to transactions that have not been committed, KTM sends TRANSACTION_NOTIFY_ROLLBACK notifications to all resource managers for the transactions that are associated with those enlistments.

The operating system deletes the object after the last handle is closed and KTM has released all its references to the object.

Transaction Objects

6/25/2019 • 2 minutes to read • Edit Online

Transaction objects represent transactions. A transactional client creates a transaction, performs some work, and the either commits or rolls back the transaction.

KTM provides a set of transaction object routines that kernel-mode transactional clients can call. KTM also provides a similar set of user-mode routines that user-mode applications can call. For more information about the user-mode routines, see the Microsoft Windows SDK.

KTM creates a transaction object when a client calls **ZwCreateTransaction**. The client can call either **ZwCommitTransaction** or **ZwRollbackTransaction** to commit or roll back the transaction.

TPS components can call **ZwOpenTransaction** to open additional handles to a transaction object.

Clients close their handles to transaction objects by calling **ZwClose**. If the last handle is closed before the transaction object has been committed, KTM sends TRANSACTION_NOTIFY_ROLLBACK notifications to all resource managers that have an enlistment for the transaction.

The operating system deletes the object after the last handle is closed and KTM has released all its references to the object.

Enlistment Objects

6/25/2019 • 2 minutes to read • Edit Online

An *enlistment object* represents a resource manager's *enlistment* to a transaction. Before a resource manager can receive notifications about a transaction's events, the resource manager must call **ZwCreateEnlistment** to create an enlistment to the transaction.

KTM provides a set of enlistment object routines that kernel-mode resource managers can call. KTM also provides a similar set of user-mode routines that user-mode applications can call. For more information about the user-mode routines, see the Microsoft Windows SDK.

KTM creates an enlistment object when a resource manager calls **ZwCreateEnlistment** to enlist in a transaction that the resource manager has received (typically from a transactional client).

TPS components can call **ZwOpenEnlistment** to open additional handles to an enlistment object. But most TPS designs do not require additional open handles.

Resource managers close their handles to enlistment objects by calling **ZwClose**. If the last handle is closed before the associated transaction object has been committed, KTM sends TRANSACTION_NOTIFY_ROLLBACK notifications to all the resource managers that have an enlistment for the transaction.

The operating system deletes the object after the last handle is closed and KTM has released all its references to the object.

Using KTM

12/5/2018 • 2 minutes to read • Edit Online

You can use the Kernel Transaction Manager (KTM) and the Common Log File System (CLFS) to create a *transaction processing system* (TPS).

This section includes the following topics:

Creating a Resource Manager

6/25/2019 • 6 minutes to read • Edit Online

Resource managers maintain each transaction's data and log the transaction's operations. If a transaction processing system (TPS) has multiple resource managers, each resource manager can participate in each transaction's commit, rollback, and recovery operations.

Each resource manager must export an interface that transactional clients can use to access the database or other resource that the resource manager maintains.

Typically, a kernel-mode resource manager must perform the following tasks in the listed order:

1. Create a log stream.

   Resource managers can use the Common Log File System (CLFS), or some other logging capability, to maintain their log streams. A call to **ClfsCreateLogFile** creates a CLFS log stream. The resource manager must use the log stream to record any information that it requires to commit, roll back, or recover transactions. In addition, KTM uses the log stream to record any internal state changes that might be necessary to recover transactions.

2. Create a transaction manager object.

   A call to **ZwCreateTransactionManager** creates a transaction manager object and connects the resource manager to an additional CLFS log stream that the resource manager specifies.

3. Recover the transaction manager state.

   A call to **ZwRecoverTransactionManager** reads the transaction manager object's log stream (which KTM maintains) and determines whether the TPS was shut down before all transactions were completed (for example, because the system crashed). KTM restores its internal state based on information in the log stream.

4. Create a resource manager object.

   A call to **ZwCreateResourceManager** creates a resource manager object and associates it with the previously created transaction manager object.

5. Recover the resource manager state.

   A call to **ZwRecoverResourceManager** causes KTM to send the resource manager TRANSACTION_NOTIFY_RECOVER notifications for any transactions that were in progress the last time that the resource manager shut down. For information about how the resource manager should respond to these notifications, see Handling Recovery Operations.

6. Receive transactions from clients.

   Typically, a client creates a transaction object and uses the resource manager's client interface to pass the transaction object's GUID to the resource manager. For example, the resource manager might provide a *CreateDataObject* routine that is similar to the one that the Understanding TPS Components topic describes.

7. Enlist in each transaction.

   A call to **ZwOpenTransaction** opens a handle to the transaction object, and then a call to **ZwCreateEnlistment** creates an enlistment for the transaction. The enlistment enables the resource

manager to receive a specified set of transaction notifications.

8. Enable reception of transaction notifications.

   The resource manager can call **ZwGetNotificationResourceManager** to obtain notifications synchronously, or it can call **TmEnableCallbacks** to register a *ResourceManagerNotification* callback routine that KTM calls whenever a notification is available.

9. Service resource access requests from clients, but do not make the changes permanent.

   After a client has created a transaction object, it typically calls the resource manager's interface to access the resource manager's resource. For example, a resource manager for a database might receive requests to read from and write to the database.

   The resource manager must record the results of the read and write operations in a CLFS log stream or other logging capability until it receives a notification that the transaction's operations will be committed, rolled back, or recovered.

10. Commit or roll back client operations.

    Eventually, the resource manager receives a notification to begin committing or rolling back the operations that the client has performed. In response, the resource manager must either make the client operations permanent or discard them. For more information about how to handle commit and rollback notifications, see Handling Transaction Operations.

    Occasionally, a resource manager might have to try to force KTM to quickly provide a commit or rollback notification, perhaps because the resource manager has determined that a device was surprise-removed. In such a case, the resource manager can call **TmRequestOutcomeEnlistment**.

11. Close the enlistment object handle.

    After the resource manager has finished processing the transaction, it must call **ZwClose** to close the enlistment object's handle

12. Close the resource manager object handle and the transaction manager object handle.

    Before the resource manager unloads, it must call **ZwClose** to close the resource manager object's handle and the transaction manager object's handle.

Steps 1 through 5 must be performed in your resource manager's initialization code. For example, if your resource manager is a kernel-mode driver, the initialization code is the driver's **DriverEntry** routine.

Steps 6 through 11 are typically performed in code that responds to requests from transactional clients.

Step 12 must be performed in your resource manager's final clean-up code, such as a kernel-mode driver's *Unload* routine.

Creating a Read-Only Enlistment

A *read-only enlistment* is an enlistment that does not receive any notifications from KTM. A resource manager can make any enlistment read-only by calling **ZwReadOnlyEnlistment**. This call causes KTM to stop delivering notifications to the resource manager.

After your resource manager has called **ZwCreateEnlistment**, it can call **ZwReadOnlyEnlistment** at any time up to the point at which it would ordinarily call **ZwPrepareComplete**.

There are two reasons why you might want your resource manager to call **ZwReadOnlyEnlistment**.

- Your resource manager has been participating in a transaction and, at some point before it receives a TRANSACTION_NOTIFY_COMMIT notification, the resource manager determines that it no longer has to

participate in the transaction's commit operation.

For example, when the resource manager receives a TRANSACTION_NOTIFY_PREPARE notification, it might determine that none of the transaction's operations have changed the resource manager's database. The resource manager can call **ZwReadOnlyEnlistment** instead of **ZwPrepareComplete** to remove itself from the transaction.

- Your resource manager never participates in any transaction's commit operation.

  For example, your resource manager might monitor data that the client sends, without modifying any stored database. In this case, your resource manager might call **ZwReadOnlyEnlistment** immediately after it has called **ZwCreateEnlistment**. In addition, you might choose to make such a resource manager *volatile*, as described in the next section of this topic.

After a resource manager has called **ZwReadOnlyEnlistment**, it can call **ZwClose** to close the enlistment handle.

Creating a Volatile-Resource Manager

A *volatile-resource manager* is a resource manager that does not maintain durable data. For example, you might create a volatile-resource manager to monitor data that the client sends, if the resource manager does not modify a durably stored database. Volatile-resource managers typically do not log transaction activity and therefore cannot perform recovery or rollback operations.

A volatile-resource manager must set the RESOURCE_MANAGER_VOLATILE flag when it calls **ZwCreateResourceManager**. If this flag is set, KTM does not log any information about the resource manager in the log stream of the associated transaction manager object.

Your resource manager can also set a TRANSACTION_MANAGER_VOLATILE flag when it calls **ZwCreateTransactionManager**. If this flag is set, KTM does not create a log stream for the transaction manager object. In addition, any additional resource managers that are connected to the transaction manager object must also be volatile and set the RESOURCE_MANAGER_VOLATILE flag.

Adding a Resource Manager to an Existing TPS

If you have to add an additional resource manager to an existing TPS, you have two choices:

- Your new resource manager calls **ZwCreateTransactionManager** to create its own transaction manager object.

  Use this choice if your resource manager does not communicate with other resource managers in the TPS.

- Your new resource manager calls **ZwOpenTransactionManager** to connect to an existing transaction manager object.

  Use this choice if your resource manager must communicate with other resource managers in the TPS. The resource manager that calls **ZwCreateTransactionManager** must share the transaction manager object's GUID, log stream name, or object name so that other resource managers can call **ZwOpenTransactionManager**. These other resource managers can call **ZwQueryInformationTransactionManager** to obtain additional information about the transaction manager object.

After you have added your resource manager to the TPS, clients that are aware of your resource manager can call the resource manager's client interface.

Creating a Transactional Client

6/25/2019 • 2 minutes to read • Edit Online

A *transactional client* is a transaction processing system (TPS) component that uses a resource manager's exported interface to access a resource, such as a database, that the resource manager supports.

Typically, the client creates a transaction, performs a set of database operations, and then commits the transaction to make the operations permanent. If the client encounters an error, it can roll back the transaction to remove the transaction's operations instead of committing the transaction.

Typically, a transactional client that uses kernel-mode KTM must perform the following tasks for each transaction:

1. Create a transaction object.

   A call to **ZwCreateTransaction** creates a transaction object, provides an object handle, and assigns an object identifier (a GUID) that the client can pass to the resource manager to identify the transaction.

2. Obtain the transaction object's identifier.

   The client can call **ZwQueryInformationTransaction** to obtain the object identifier.

3. Pass the transaction object's identifier to a resource manager.

   The client typically calls the resource manager's exported interface to open a communication path to the resource manager and to associate the path with the transaction. For example, the resource manager might provide a *CreateDataObject* routine that is similar to the one that the Understanding TPS Components topic describes.

4. Perform operations to be included in the transaction.

   Typically, the client calls the resource manager's interface to access the resource manager's resource. For example, the client of a database manager might read from and write to the database.

5. Commit or roll back the transaction.

   If all the resource operations succeed, the client must call **ZwCommitTransaction** to make the operations permanent. If an operation fails, the client must call **ZwRollbackTransaction** instead of **ZwCommitTransaction**. For example, if the client of a database manager determines that one of a series of write operations failed, the client must call **ZwRollbackTransaction** so that none of the write operations become permanent.

   Clients can call **ZwCommitTransaction** and **ZwRollbackTransaction** either synchronously or asynchronously. If clients call these routines synchronously, the routines do not return until the commit or rollback operation is complete.

   For more information about how to commit and roll back transactions, see Handling Transaction Operations.

6. Close the transaction object handle.

   After the client has finished processing the transaction, it must call **ZwClose** to close the transaction object's handle

A TPS might include more than one resource manager. If a client's transaction includes operations on multiple resources, such as two databases that two resource managers support, the client typically does the following:

1. Creates a single transaction object for each transaction.
2. Passes the transaction object's identifier to each resource manager.
3. Performs operations on each database by calling each resource manager's interface.
4. Commits the transaction if all operations completed without errors, or rolls back the transaction if an error was detected.

If your TPS includes a *superior transaction manager*, transactional clients typically do not call KTM. For more information about superior transaction managers and their clients, see Creating a Superior Transaction Manager.

Transactional clients can call **ZwSetInformationTransaction** to set transaction-specific information. For example, a client can set a time-out value for the transaction or supply a descriptive character string. Clients can call **ZwQueryInformationTransaction** to retrieve information about a transaction. For example, a client can call this routine to determine whether a transaction has been committed or rolled back.

Creating a Superior Transaction Manager

6/25/2019 • 2 minutes to read • Edit Online

In KTM, a *superior transaction manager* is a resource manager that creates superior enlistments for the transactions that it participates in. A *superior enlistment* is an enlistment that grants the resource manager the ability to coordinate the commit operation for the enlistment's transaction. In other words, either a transactional client or the superior transaction manager can start the pre-prepare/prepare/commit sequence for the transaction.

After a resource manager has created a superior enlistment for a transaction, KTM rejects all calls to **ZwCommitTransaction** for the transaction. Therefore, transactional clients cannot commit such a transaction. Instead, the resource manager that created the superior enlistment must call **ZwPrePrepareEnlistment**, **ZwPrepareEnlistment**, and **ZwCommitEnlistment**.

When to Create a Superior Transaction Manager

Suppose that you want to integrate a transaction processing system (TPS) component with KTM, but the component contains its own non-KTM transaction management capabilities that clients can call. In such a situation, you might have to create a superior transaction manager.

For example, suppose that your component provides its own interfaces that clients use to create and commit transactions. Because your component's clients do not call KTM to create or commit transactions, your component must become a superior transaction manager when you integrate it into a KTM-based TPS.

How to Create a Superior Transaction Manager

If you want your component to be a superior transaction manager, it must do the following:

1. Call **ZwCreateResourceManager** to register as a resource manager.
2. Call **ZwCreateTransaction** every time that a client of your component creates a transaction by using your component's client interface.
3. Call **ZwCreateEnlistment**, setting the ENLISTMENT_SUPERIOR flag, and specifying both the ENLISTMENT_SUPERIOR_RIGHTS and ENLISTMENT_SUBORDINATE_RIGHTS access flags.
4. Call **ZwPrePrepareEnlistment**, **ZwPrepareEnlistment**, and **ZwCommitEnlistment** when your component's client calls your component's client interface to commit the transaction.

KTM permits only one superior enlistment per transaction. Other resource managers can create additional enlistments. These enlistments are called *subordinate enlistments* because they cannot initiate the commit operation.

To roll back a superior enlistment, your superior transaction manager calls **ZwRollbackEnlistment**.

To recover a superior enlistment, your superior transaction manager calls **ZwRecoverEnlistment**.

When a superior transaction manager commits, rolls back, or recovers a transaction, KTM sends transaction notifications to all subordinate enlistments so that they can participate.

A TPS that includes a superior transaction manager cannot use single-phase commit operations.

During a recovery operation, if KTM cannot determine the outcome of a transaction, it sends a TRANSACTION_NOTIFY_RECOVER_QUERY notification to the superior transaction manager. In response, the superior transaction manager must call **ZwCommitEnlistment** if the transaction can be committed or **ZwRollbackEnlistment** if the transaction should be rolled back. If the superior transaction manager cannot

determine the outcome of a transaction, it should not respond to the TRANSACTION_NOTIFY_RECOVER_QUERY notification until it can determine an outcome.

Handling Transaction Operations

12/5/2018 • 2 minutes to read • Edit Online

Resource managers must handle three transaction operations: *commit*, *rollback*, and *recovery*.

To *commit a transaction*, a resource manager makes all changes to a transaction's data permanent and visible to other transactions.

To *roll back a transaction*, a resource manager removes all changes to a transaction's data. The resource manager must restore the data to the state that it was in before the transaction was created.

To *recover a transaction*, a resource manager restores a transaction's data to a known good state after a system crash or another unexpected event.

This section contains the following topics:

Handling Commit Operations

6/25/2019 • 4 minutes to read • Edit Online

There are two types of commit operations: *single-phase commit* and *multi-phase commit*. A single-phase commit operation consists of a single notification that resource managers must respond to, while a multi-phase commit operation includes additional notifications for preparation steps.

A single-phase commit operation is simpler to implement. It is appropriate for transaction processing systems (TPSs) that have one of the following characteristics:

- A single resource manager.
- Multiple resource managers, all but one of which are read-only and do not participate in the commit operation.

A multi-phase commit operation is necessary if multiple resource managers participate in the commit operation.

Single-Phase Commit Operations

If you want your TPS to support single-phase commit operations, one (and only one) resource manager must register to receive TRANSACTION_NOTIFY_SINGLE_PHASE_COMMIT notifications for its enlistments. All other resource managers must be read-only.

A TPS that includes a superior transaction manager cannot use single-phase commit.

If a resource manager has registered to receive TRANSACTION_NOTIFY_SINGLE_PHASE_COMMIT notifications, KTM sends this kind of notification when a transactional client calls **ZwCommitTransaction**.

When the resource manager receives a TRANSACTION_NOTIFY_SINGLE_PHASE_COMMIT notification for a transaction, it can either commit the transaction or reject single-phase commit.

To commit the transaction, the resource manager must do the following:

1. Flush any data that it is holding in a non-permanent cache (in-memory storage), such as the CLFS marshalling area for a CLFS log stream.

   The resource manager must move the data from the cache to a durable storage medium. For example, a resource manager that is using CLFS can call **ClfsFlushBuffers**.

2. Make all data changes permanent and public (that is, visible outside the resource manager's scope).

3. Call **ZwCommitComplete**.

After calling **ZwCommitComplete**, the resource manager should call **ZwClose** to close the enlistment handle.

To reject a single-phase commit operation for the transaction, the resource manager can call **ZwSinglePhaseReject**. If the resource manager calls **ZwSinglePhaseReject**, KTM immediately changes the commit operation from single-phase to multi-phase.

If other resource managers enlist in the same transaction, they must be read-only. However, they must register to receive the TRANSACTION_NOTIFY_RM_DISCONNECTED notification, which they receive if the resource manager that is handling the single-phase commit operation closes the enlistment handle without indicating that it has committed or rolled back the transaction.

Multi-Phase Commit Operations

A multi-phase commit operation begins when one of the following events happens:

- A transactional client calls **ZwCommitTransaction**, and no resource managers have registered to receive TRANSACTION_NOTIFY_SINGLE_PHASE_COMMIT notifications.
- A resource manager calls **ZwSinglePhaseReject** after it has received a TRANSACTION_NOTIFY_SINGLE_PHASE_COMMIT notification.
- A superior transaction manager calls **ZwPrePrepareEnlistment**.

Multi-phase commit operations consist of three sequential phases: *pre-prepare*, *prepare*, and *commit*.

Pre-Prepare Phase

The pre-prepare phase (also known as *phase zero*) of the commit operation begins when KTM sends a TRANSACTION_NOTIFY_PREPREPARE notification to all resource managers. KTM sends this notification if no resource managers support a single-phase commit operation for the transaction, or if a superior transaction manager calls **ZwPrePrepareEnlistment**.

When each resource manager receives the TRANSACTION_NOTIFY_PREPREPARE notification, it must do the following:

1. Flush any data that it is holding in a non-permanent cache (in-memory storage), such as the CLFS marshalling area for a CLFS log stream.

   The resource manager must move the data from the cache to a durable storage medium. For example, a resource manager that is using CLFS can call **ClfsFlushBuffers**.

2. Call **ZwPrePrepareComplete**.

After a resource manager has called **ZwPreprepareComplete**, it can continue to receive and service client requests. But the resource manager must treat all data modifications as cache pass-through operations that are immediately written to a durable storage medium.

If a resource manager encounters an error while it is processing a TRANSACTION_NOTIFY_PREPREPARE notification, it should call **ZwRollbackEnlistment** to roll back the transaction.

Prepare Phase

The prepare phase (also known as *phase one*) of the commit operation begins when KTM sends a TRANSACTION_NOTIFY_PREPARE notification to all resource managers. KTM sends this notification after TRANSACTION_NOTIFY_PREPREPARE if no resource managers support single-phase commit or if a superior transaction manager calls **ZwPrepareEnlistment**.

When each resource manager receives the TRANSACTION_NOTIFY_PREPARE notification, it must do the following:

1. Stop servicing client requests and report any client subsequent requests as client errors.
2. Make sure that all data has been moved to durable storage.
3. Call **ZwPrepareComplete**.

If a resource manager encounters an error while it is processing a TRANSACTION_NOTIFY_PREPARE notification, it should call **ZwRollbackEnlistment** to roll back the transaction. However, the resource manager cannot roll back the transaction after it has called **ZwPrepareComplete**.

Commit Phase

The commit phase (also known as *phase two*) of the commit operation begins when KTM sends a TRANSACTION_NOTIFY_COMMIT notification to all resource managers. KTM sends this notification after TRANSACTION_NOTIFY_PREPARE if no resource managers support single-phase commit or if a superior

transaction manager calls **ZwCommitEnlistment**.

When each resource manager receives the TRANSACTION_NOTIFY_COMMIT notification, it must do the following:

1. Make all data changes permanent and public (that is, visible to other transactions).

   Typically, a resource manager makes changes permanent and public by copying the transaction's saved data from the log stream to the database's public, permanent storage. For more information about how to use log streams, see Using Log Streams with KTM.

2. Call **ZwCommitComplete**.

After the resource manager calls **ZwCommitComplete**, it should call **ZwClose** to close the enlistment handle.

If a resource manager encounters an error while it is processing a TRANSACTION_NOTIFY_COMMIT notification, it should shut itself down. The next time that the operating system reloads the resource manager, the resource manager's recovery process should restore the transaction to a state that was known to be good before the error occurred.

Handling Rollback Operations

6/25/2019 • 2 minutes to read • Edit Online

A resource manager, a transactional client, or KTM can roll back a transaction if it determines that the transaction must not be committed (typically because an error has been detected).

To roll back a transaction, a resource manager can call **ZwRollbackEnlistment**. After the resource manager has called **ZwCreateEnlistment** to enlist in a transaction, it can roll back the transaction at any time before it calls **ZwPrepareComplete**.

Transactional clients can roll back their transactions by calling **ZwRollbackTransaction**. After a transactional client has called **ZwCreateTransaction** to create a transaction, it can roll back the transaction at any time before it calls **ZwCommitTransaction**.

In addition, a transactional client can set a time-out value for a transaction by calling **ZwSetInformationTransaction**. KTM rolls back the transaction if it has not been committed by the specified amount of time.

When a call to **ZwRollbackEnlistment** or **ZwRollbackTransaction** is made, or when a time-out value is exceeded, KTM sends a TRANSACTION_NOTIFY_ROLLBACK notification to all resource managers.

When each resource manager receives a TRANSACTION_NOTIFY_ROLLBACK notification, it must do the following:

1. Restore the transaction's data to the state that it was in before the resource manager enlisted in the transaction.

   Typically, a resource manager restores the transaction's data by copying the transaction's saved initial data from the log stream to the database's public, permanent storage. For more information about how to use log streams, see Using Log Streams with KTM.

2. Call **ZwRollbackComplete**.

After calling **ZwRollbackComplete**, the resource manager should call **ZwClose** to close the enlistment handle.

If a resource manager initiated the rollback operation, it must use its client interface to inform the client that the transaction failed.

Handling Recovery Operations

6/25/2019 • 2 minutes to read • Edit Online

In a *recovery* operation, a transaction processing system (TPS) tries to recover its state from the information that is in log streams. After a recovery operation is complete, all transactions should be in a committed or rolled back state, and all resource data should be in a known good state.

Sometimes a TPS stops before all its transactions have finished. For example, the operating system might crash. Therefore, resource managers must initiate recovery operations whenever they start to run. The recovery operation tries to determine whether any transactions are incomplete. If incomplete transactions are found in the log, the recovery operation tries to commit or roll back those transactions.

For a KTM-based TPS, each recovery operation consists of two steps. The first step involves recovering information from the transaction manager object's log stream. The second step involves recovering information from the resource manager's log stream.

A TPS can recover to the end of all log streams or, if its resource managers maintain virtual clock values, it can recover up to a specified clock value.

Recovering Information from a Transaction Manager Object's Log Stream

Immediately after a resource manager calls **ZwCreateTransactionManager** or **ZwOpenTransactionManager**, it must call **ZwRecoverTransactionManager**. The **ZwRecoverTransactionManager** routine reads the log stream that belongs to the transaction manager object. This routine reconstructs the state of the transaction manager object (including all transactions, enlistments, and resource managers) from the recovery information that is in the log stream, beginning at the last restart area that KTM created and ending at the stream's end.

To recover from the last restart area up to a specified virtual clock value, the resource manager can call **ZwRollforwardTransactionManager** instead of **ZwRecoverTransactionManager**.

Recovering Information from a Resource Manager's Log Stream

Immediately after a resource manager calls **ZwCreateResourceManager** or **ZwOpenResourceManager**, it must call **ZwRecoverResourceManager**. The **ZwRecoverResourceManager** routine tries to recover the transactions that are associated with each of the resource manager's enlistments.

When a resource manager calls **ZwRecoverResourceManager**, KTM sends TRANSACTION_NOTIFY_RECOVER notifications for each of the resource manager's enlistments. The resource manager must call **ZwRecoverEnlistment** every time that it receives one of the TRANSACTION_NOTIFY_RECOVER notifications.

When the resource manager calls **ZwRecoverEnlistment**, KTM sends one of the following notifications:

- TRANSACTION_NOTIFY_COMMIT

  The resource manager must use information in its log stream to commit the transaction and then must call **ZwCommitComplete**.

- TRANSACTION_NOTIFY_ROLLBACK

  The resource manager must use information in its log stream to roll back the transaction and then must call **ZwRollbackComplete**.

- TRANSACTION_NOTIFY_INDOUBT

  KTM has not determined the state of the transaction and will send a commit or rollback notification later.

Typically, KTM sends a TRANSACTION_NOTIFY_COMMIT notification if it determines that all resource managers called **ZwPrepareComplete** before the TPS stopped and restarted. KTM sends a TRANSACTION_NOTIFY_ROLLBACK notification if it determines that one or more resource managers did not call **ZwPrepareComplete**.

After KTM has sent a TRANSACTION_NOTIFY_RECOVER notification for each enlistment, it sends a TRANSACTION_NOTIFY_LAST_RECOVER notification.

If your resource manager called **ZwRollforwardTransactionManager** instead of **ZwRecoverTransactionManager**, it must recover only up to the virtual clock value that it specified to **ZwRollforwardTransactionManager**.

Resource managers can call **ZwSetInformationEnlistment** to set customized recovery information. KTM saves this information and writes it to the log stream, but KTM does not try to interpret the information. The resource manager can retrieve the recovery information at any time by calling **ZwQueryInformationEnlistment**.

Superior transaction managers sometimes receive TRANSACTION_NOTIFY_RECOVER_QUERY notifications during a recover operation.

Transaction Notifications

6/25/2019 • 4 minutes to read • Edit Online

KTM provides a notification queue for each resource manager. KTM delivers notifications to a resource manager by putting them in the resource manager's queue.

A resource manager can retrieve notifications from its queue either synchronously or asynchronously.

- To retrieve notifications synchronously, the resource manager can repeatedly call **ZwGetNotificationResourceManager**.
- To receive notifications asynchronously, the resource manager can call **TmEnableCallbacks** to set up a callback routine. KTM calls the callback routine every time that it puts a notification in the resource manager's queue.

When a resource manager calls **ZwCreateEnlistment** to create an enlistment for a transaction, the resource manager specifies the types of notifications that it should receive. Resource managers receive only notifications that they register to receive.

The notification constants are defined in Ktmtypes.h. Notification constant names have a format of TRANSACTION_NOTIFY_*Xxx*.

The rest of this topic lists all the notification constants that Ktmtypes.h defines and divides them into three groups:

- Notifications that resource managers can receive
- Notifications that superior transaction managers can receive
- Notification constants that are defined but currently not used

Notifications for Resource Managers

All resource managers must register to receive TRANSACTION_NOTIFY_PREPREPARE, TRANSACTION_NOTIFY_PREPARE, and TRANSACTION_NOTIFY_COMMIT notifications, even if they subsequently call **ZwReadOnlyEnlistment** to mark an enlistment as read-only.

Resource managers can support TRANSACTION_NOTIFY_SINGLE_PHASE_COMMIT, but they must also support the multi-phase pre-prepare, prepare, and commit notifications.

The following list contains all the notifications that resource managers can receive:

TRANSACTION_NOTIFY_PREPREPARE
When sent: A client calls **ZwCommitTransaction** and no resource manager supports single-phase commit, or if a superior transaction manager calls **ZwPrePrepareEnlistment**.

Received by: Resource managers.

Recipient's required action: Perform pre-prepare operations and then call **ZwPrePrepareComplete**. (For more information about pre-prepare operations, see Handling Commit Operations.)

Restrictions: The resource manager must also support TRANSACTION_NOTIFY_PREPARE and TRANSACTION_NOTIFY_COMMIT.

TRANSACTION_NOTIFY_PREPARE
When sent: After TRANSACTION_NOTIFY_PREPREPARE if a client calls **ZwCommitTransaction** and no resource manager supports single-phase commit, or if a superior transaction manager calls

ZwPrepareEnlistment.

Received by: Resource managers.

Recipient's required action: Perform prepare operations and then call **ZwPrepareComplete**. (For more information about prepare operations, see Handling Commit Operations.)

Restrictions: The resource manager must also support TRANSACTION_NOTIFY_PREPREPARE and TRANSACTION_NOTIFY_COMMIT.

TRANSACTION_NOTIFY_COMMIT
When sent: After TRANSACTION_NOTIFY_PREPARE if a client calls **ZwCommitTransaction** and no resource manager supports single-phase commit, or if a superior transaction manager calls **ZwCommitEnlistment**.

Received by: Resource managers.

Recipient's required action: Perform commit operations and then call **ZwCommitComplete**. (For more information about commit operations, see Handling Commit Operations.)

Restrictions: The resource manager must also support TRANSACTION_NOTIFY_PREPREPARE and TRANSACTION_NOTIFY_PREPARE.

TRANSACTION_NOTIFY_SINGLE_PHASE_COMMIT
When sent: A client calls **ZwCommitTransaction** and a resource manager supports single-phase commit operations.

Received by: Resource managers.

Recipient's required action: Either commit the transaction or call **ZwSinglePhaseReject**. (For more information about single-phase commit operations, see Handling Commit Operations.)

Restrictions: The resource manager must also support TRANSACTION_NOTIFY_PREPREPARE, TRANSACTION_NOTIFY_PREPARE, and TRANSACTION_NOTIFY_COMMIT.

TRANSACTION_NOTIFY_ROLLBACK
When sent: A client calls **ZwRollbackTransaction**, a superior transaction manager calls **ZwRollbackEnlistment**, or KTM detects an error (such as a failed write to the log stream).

Received by: Both resource managers and superior transaction managers.

Recipient's required action: Perform any operations that are needed to roll back the transaction's data, and then call **ZwRollbackComplete**. (For more information about rollback operations, see Handling Rollback Operations.)

Restrictions: All resource managers and superior transaction managers must support TRANSACTION_NOTIFY_ROLLBACK.

TRANSACTION_NOTIFY_RECOVER
When sent: A resource manager calls **ZwRecoverResourceManager**.

Received by: Resource managers.

Recipient's required action: The resource manager must call **ZwRecoverEnlistment**. (For more information about recovery operations, see Handling Recovery Operations.)

Restrictions: None.

TRANSACTION_NOTIFY_LAST_RECOVER
When sent: After KTM has sent the last TRANSACTION_NOTIFY_RECOVER for a resource manager's enlistments.

Received by: Resource managers.

Recipient's required action: End the recovery operation. (For more information about recovery operations, see Handling Recovery Operations.)

Restrictions: None.

TRANSACTION_NOTIFY_INDOUBT
When sent: After a resource manager calls **ZwRecoverEnlistment**, if KTM cannot determine whether the transaction should be committed or rolled back (typically because the TPS has a superior transaction manager that is unavailable).

Received by: Resource managers.

Recipient's required action: Do nothing until KTM sends TRANSACTION_NOTIFY_COMMIT or TRANSACTION_NOTIFY_ROLLBACK.

Restrictions: None.

TRANSACTION_NOTIFY_RM_DISCONNECTED
When sent: The resource manager that is handling a single-phase commit operation closes the enlistment handle without indicating that it has committed or rolled back the transaction.

Received by: Resource managers and superior transaction managers that have enlistments for the transaction.

Recipient's required action: Transaction-specific cleanup operations. Typically, this notification is useful to read-only resource managers.

Restrictions: None.

Notifications for Superior Transaction Managers

Superior transaction managers can receive the following notifications:

TRANSACTION_NOTIFY_ROLLBACK
See earlier description.

TRANSACTION_NOTIFY_RM_DISCONNECTED
See earlier description.

TRANSACTION_NOTIFY_PREPREPARE_COMPLETE
When sent: After all resource managers have received TRANSACTION_NOTIFY_PREPREPARE and responded by calling **ZwPrePrepareComplete**.

Received by: Superior transaction managers.

Recipient's required action: The superior transaction manager should call **ZwPrepareEnlistment**.

TRANSACTION_NOTIFY_PREPARE_COMPLETE
When sent: After all resource managers have received TRANSACTION_NOTIFY_PREPARE and responded by calling **ZwPrepareComplete**.

Received by: Superior transaction managers.

Recipient's required action: The superior transaction manager should call **ZwCommitEnlistment**.

TRANSACTION_NOTIFY_COMMIT_COMPLETE
When sent: After all resource managers have received TRANSACTION_NOTIFY_COMMIT and responded by calling **ZwCommitComplete**.

Received by: Superior transaction managers.

Recipient's required action: Transaction cleanup operations.

TRANSACTION_NOTIFY_ROLLBACK_COMPLETE
When sent: After all resource managers have received TRANSACTION_NOTIFY_ROLLBACK and responded by calling **ZwRollbackComplete**.

Received by: Superior transaction managers.

Recipient's required action: Transaction cleanup operations.

TRANSACTION_NOTIFY_RECOVER_QUERY
When sent: A superior transaction manager calls **ZwRecoverResourceManager**.

Received by: Superior transaction managers.

Recipient's required action: The superior transaction manager must call either **ZwCommitEnlistment** or **ZwRollbackEnlistment** for the enlistment.

TRANSACTION_NOTIFY_COMMIT_REQUEST
When sent: A client calls **ZwCommitTransaction**. If a superior transaction manager has registered for this notification for an enlistment, KTM sends TRANSACTION_NOTIFY_COMMIT_REQUEST to the superior transaction manager **instead of** sending TRANSACTION_NOTIFY_COMMIT to the resource managers.

Received by: Superior transaction managers.

Recipient's required action: The superior transaction manager calls **ZwCommitEnlistment**.

TRANSACTION_NOTIFY_REQUEST_OUTCOME
When sent: A resource manager calls **TmRequestOutcomeEnlistment** while the transaction is in its prepared state.

Received by: Superior transaction managers.

Recipient's required action: The superior transaction manager must call **ZwCommitEnlistment** or **ZwRollbackEnlistment**.

Unused Notifications

The following notifications are defined in Ktmtypes.h, but KTM currently does not support them:

TRANSACTION_NOTIFY_DELEGATE_COMMIT

TRANSACTION_NOTIFY_ENLIST_MASK

TRANSACTION_NOTIFY_ENLIST_PREPREPARE

TRANSACTION_NOTIFY_MARSHAL

TRANSACTION_NOTIFY_PROMOTE

TRANSACTION_NOTIFY_PROMOTE_NEW

TRANSACTION_NOTIFY_PROPAGATE_PULL

TRANSACTION_NOTIFY_PROPAGATE_PUSH

TRANSACTION_NOTIFY_TM_ONLINE

TRANSACTION_NOTIFY_COMMIT_FINALIZE

Using Log Streams with KTM

6/25/2019 • 3 minutes to read • Edit Online

KTM-based transaction processing systems (TPSs) should log transaction activity by using the Common Log File System (CLFS). KTM creates a log stream for each transaction manager object. Each resource manager should create its own log stream.

Creating Log Streams for Transaction Manager Objects

When your resource manager calls **ZwCreateTransactionManager**, you must specify the name of a CLFS log stream. If the specified stream does not exist, KTM creates it. If the stream already exists, **ZwCreateTransactionManager** reopens it. KTM assigns this log stream to the transaction manager object.

KTM uses the transaction manager object's log stream to record internal state information about the transaction manager object and all resource manager objects, transaction objects, and enlistment objects that are associated with the transaction manager object. If transactional operations are interrupted before they are complete, KTM can use the information in the log to determine whether to commit or roll back the transactions.

KTM does not record the transaction data that resource managers receive from or send to clients. Resource managers must use their own log streams to record this information.

Resource managers can call **ZwQueryInformationTransactionManager** to obtain information about a transaction manager object's log stream, such as the log stream's path name or the GUID that KTM assigns to the stream.

Creating Log Streams for Resource Managers

In its initialization code, each resource manager should call **ClfsCreateLogFile** to create its own log stream. Each resource manager should use its stream to record all the information about transactions that it requires to commit, roll back, or recover the transaction's data.

KTM and all resource managers of a TPS can use a single log file, but each TPS component must use a different stream within the log file. For information about how to specify individual streams within a log file, see **ClfsCreateLogFile**.

Periodically, KTM creates a restart area in the transaction manager's log stream. When KTM performs a recovery operation, it reads the last restart area to recover the state of objects that were open before the system shut down. Similarly, your resource manager should periodically create restart areas in its log stream. For example, your resource manager might create a restart area every time that a transactional operation is completed.

For more information about restart areas in CLFS log streams, see Reading Restart Records from a CLFS Stream. Also, see the **ClfsWriteRestartArea**, **ClfsReadRestartArea**, and **ClfsReadPreviousRestartArea** routines.

Using Log Streams for Recovery

After your resource manager calls **ZwCreateTransactionManager**, it must call **ZwRecoverTransactionManager**. The **ZwRecoverTransactionManager** routine reads the transaction manager object's log stream to recover the state of the TPS to a known good point. If the computer shut down properly—or did not shut down—after the resource manager was last loaded, the log stream contains minimal information. If a system crash has occurred, the log stream contains enough recovery information to restore all the transactions to a known state.

After your resource manager calls **ZwCreateResourceManager**, it must call **ZwRecoverResourceManager**. The **ZwRecoverResourceManager** routine tries to recover the transactions that are associated with each of the

resource manager's enlistments. For more information about how to recover a resource manager's transactions, see Handling Recovery Operations.

Storing Transaction Data

Resource managers that use CLFS log streams should store transaction data in CLFS marshalling areas. CLFS periodically moves the data from the log stream's marshalling area to a permanent storage medium. To log an operation that modifies data, a resource manager might do the following:

1. Copy the original data, before the write operation modifies it, to the marshalling area.
2. Perform the operation on a copy of the data without modifying the database's permanent storage medium.
3. Copy the new data to the marshalling area.

If the resource manager receives a rollback notification, it can restore the original data from the log stream. If it receives a commit notification, the resource manager can copy the modified data from the log stream to the database's permanent storage medium.

Resource managers can also use the **ZwSetInformationEnlistment** routine to store recovery information in an enlistment object. KTM saves this information in its log stream and reads it from the log stream during recovery operations. Therefore, a resource manager can obtain this recovery information at any time by calling **ZwQueryInformationEnlistment**.

Using Virtual Clock Values

6/25/2019 • 2 minutes to read • Edit Online

KTM provides a virtual clock for each transaction manager object. When a resource manager calls **ZwCreateTransactionManager**, KTM sets the object's virtual clock value to 1. KTM increments the virtual clock value every time that a commit operation begins. Whenever KTM writes to its log stream, it includes the current virtual clock value in the log record.

When a resource manager calls **ZwRecoverTransactionManager**, KTM reads log stream records up to the end of the stream, and it sets the transaction manager object's virtual clock value to the last value that it finds in the object's log stream.

When a resource manager calls **ZwRollforwardTransactionManager**, KTM reads log stream records up to the specified clock value, and it sets the transaction manager object's virtual clock value to the specified clock value.

KTM enables resource managers and superior transaction managers to modify a transaction manager object's virtual clock value, but they typically do not have to modify the clock value.

When to Modify Virtual Clock Values

Typically, your transaction processing system (TPS) does not have to modify virtual clock values unless the components in your TPS are trying to synchronize multiple log streams.

For example, suppose that your TPS contains multiple resource managers that communicate with each other during pre-prepare/prepare/commit sequences. Also suppose that each resource manager creates a transaction manager object that has a unique log stream. To make sure that KTM restores the state of all resource managers to the same point in time during a recovery operation, these resource managers might use the following steps:

- When one resource manager communicates with another, it passes the most recent virtual clock value that it has received from either KTM or yet another resource manager.
- Whenever a resource manager calls a KTM routine that accepts a virtual clock value (see the following section in this topic), it passes the highest clock value that it has received from KTM or another resource manager.
- Each resource manager writes virtual clock values into its log stream and uses those values when it performs rollback or recovery operations.

These steps cause the virtual clock values that KTM stores for each transaction manager object to almost or exactly match. Therefore, when a recovery operation causes KTM to read its log streams, or when a rollback operation causes the resource managers to read their log streams, the recovery or rollback is based on synchronized log streams.

How to Modify Virtual Clock Values

Resource managers can modify the virtual clock value by passing a new value to **ZwPrePrepareComplete**, **ZwPrepareComplete**, **ZwCommitComplete**, **ZwRollbackComplete**, **ZwReadOnlyEnlistment**, or **ZwSinglePhaseReject**.

Superior transaction managers can modify the virtual clock value by passing a new value to **ZwPrePrepareEnlistment**, **ZwPrepareEnlistment**, **ZwCommitEnlistment**, or **ZwReadOnlyEnlistment**.

In addition, a resource manager or superior transaction manager that uses a **ResourceManagerNotification** callback routine can modify the virtual clock value that the callback routine receives. KTM then saves the updated value.

If a resource manager or superior transaction manager passes a new clock value to KTM, KTM saves the new value only if it is greater than the current clock value. Otherwise, KTM keeps the current clock value.

Resource managers and superior transaction managers can obtain a transaction manager object's virtual clock value by calling the **ZwQueryInformationTransactionManager** routine.

Using TmXxx Routines

6/25/2019 • 2 minutes to read • Edit Online

Most KTM routines use a naming format of **Zw*Xxx***. These routines are handle-based. That is, at least one of their input or output parameters is a handle to a KTM object.

KTM also provides a smaller number of routines that use a naming format of **Tm*Xxx***. These routines are pointer-based. At least one of their input or output parameters is a pointer to a KTM object.

Some **Tm*Xxx*** routines duplicate **Zw*Xxx*** routines. Other **Tm*Xxx*** routines do not have **Zw*Xxx*** equivalents.

In most cases, you should use the **Zw*Xxx*** routines. But you should use **Tm*Xxx*** routines in the following situations:

- Your resource manager uses the **ResourceManagerNotification** callback routine, which provides a pointer to an enlistment object instead of a handle.

  You can pass the enlistment object pointer to the enlistment object's **Tm*Xxx*** routines.

- Your transaction processing system (TPS) component performs many rapid calls to KTM, which potentially causes system performance to be too slow.

  In this case, your component can call **ObReferenceObjectByHandle** to convert each KTM object handle to a pointer, save the pointer, and then pass the pointer to **Tm*Xxx*** routines. This conversion eliminates the need for KTM to convert each handle to a pointer internally every time that a **Zw*Xxx*** routine is called.

  Each call to **ObReferenceObectByHandle** should include an access mask that contains appropriate KTM-defined flags. These flags are described on the reference pages for KTM's create and open routines.

  When your component has finished using the KTM object, it must dereference the object by calling either **ObDereferenceObjectDeferDelete** or **ObDereferenceObject**.

  - You must use **ObDereferenceObjectDeferDelete** if your component, or any other component in your driver stack, is holding any system-provided locks, such as spin locks, mutex objects, or fast mutexes.

  - You can use **ObDereferenceObject** if you are sure that no component on your driver stack holds system-provided locks.

  Deadlocks can occur if your component calls **ObDereferenceObject** while holding locks, because KTM might also be holding locks for the object namespace. Also, your component can call **TmGetTransactionId** to quickly obtain a transaction's identifier more efficiently than calling **ZwQueryInformationTransaction**.

- You must have a capability that a **Zw*Xxx*** routine does not provide.

  Specifically, a resource manager can call the following routines:

  - **TmEnableCallbacks** to enable asynchronous delivery of notifications by a callback routine.
  - **TmReferenceEnlistmentKey** and **TmDereferenceEnlistmentKey** to increment or decrement an enlistment object's key reference count.
  - **TmRequestOutcomeEnlistment** to request an immediate commit or rollback notification for an enlistment.
  - **TmIsTransactionActive** to determine whether a transaction is in its active state.

Dynamic Hardware Partitioning Techniques

12/5/2018 • 2 minutes to read • Edit Online

Changing the hardware configuration of a server while the server is running is known as *dynamic hardware partitioning*. If you want to run your device drivers on servers that support dynamic hardware partitioning, your drivers must support dynamic changes to the hardware configuration of the server.

This section includes the following topics:

Introduction to Dynamic Hardware Partitioning

12/5/2018 • 4 minutes to read • Edit Online

A *hardware partitionable server* is a server that can be configured into one or more isolated *hardware partitions*. Each hardware partition runs an independent instance of the operating system. You can assign each of the server's hardware resources to each of the various hardware partitions in whatever configuration is appropriate for the server's application. The hardware resources that are assigned to a particular hardware partition are isolated from the other hardware partitions in the server.

A hardware partition consists of one or more *partition units*. A partition unit is the smallest unit of hardware that you can assign to a hardware partition. A partition unit can be a processor, a memory module, or an I/O host bridge. Typically, processors and memory modules are plugged into sockets that can be powered on or off independently.

A hardware partitionable server can be one of two types: *statically partitionable* or *dynamically partitionable*. On a statically partitionable server, you cannot change the configuration of partition units that are assigned to each hardware partition while the server is running. To change the configuration, you must shut down and restart the server computer. Microsoft Windows Server 2000 and later versions of the Windows Server operating system support statically partitionable servers.

On a dynamically partitionable server, you can change the configuration of the partition units that are assigned to each hardware partition while the server is running. This is known as *dynamic hardware partitioning*. If the operating system that is running on a hardware partition supports dynamic hardware partitioning, you can add, replace, or remove partition units without restarting the operating system. Depending on the capabilities of the operating system, you can perform one or more of the following dynamic hardware partitioning operations:

Hot add
Adding a partition unit to a running hardware partition.

Hot remove
Removing a partition unit from a running hardware partition.

Hot replace
Replacing a partition unit with an identical replacement partition unit that is already present in the server computer. A hot replace operation is a single operation that differs from a hot remove operation followed by a hot add operation.

Windows Server 2003 with Service Pack 1 (SP1) supports hot add operations for memory modules on x86-based, x64-based, and Itanium-based servers. Windows Server 2003 SP1 does not support hot remove or hot replace operations.

Starting with Windows Server 2008, the operating system supports hot add operations for processors, memory modules, and I/O host bridges, and hot replace operations for processors and memory modules on x64-based and Itanium-based server computers. The operating system also supports hot add operations for memory modules on x86-based server computers. The operating system does not support hot remove operations.

The following table summarizes the dynamic hardware partitioning support that is included in each version of Windows Server.

	WINDOWS SERVER 2003 WITH SP1	WINDOWS SERVER 2008 AND LATER VERSIONS OF WINDOWS SERVER ON X86-BASED SERVERS	WINDOWS SERVER 2008 AND LATER VERSIONS OF WINDOWS SERVER ON X64-BASED AND ITANIUM-BASED SERVERS
hot add	memory modules	memory modules	processors, memory modules, I/O host bridges
hot remove			
hot replace			processors, memory modules

We suggest that you consider the following guidelines when you develop your device drivers:

- You should understand dynamic hardware partitioning because certain assumptions about the hardware configuration of a server computer are not valid on dynamically partitionable servers. Device drivers that are not designed to accommodate dynamic hardware partitioning could cause data corruption or cause the operating system to generate a bug check if they are run on a dynamically partitionable server.
- You should consider the critical issues that are identified for dynamic hardware partitioning, even if you are not developing device drivers for server computers.
- You should review and update all the device drivers that you are developing for servers that run Windows Server 2008 and later versions of Windows Server. Device drivers can register with the operating system to be notified of changes to the hardware configuration. When the device drivers are notified about a change to the hardware configuration, they can respond to the change as required for safe and optimal operation. This ensures that the drivers function correctly on dynamically partitionable servers.

Drivers that you develop for Windows XP and later versions of Windows that correctly participate in resource rebalancing and do not make any assumptions about the number of processors, the processor affinity mask, or the amount of physical memory, will continue to operate correctly on a dynamically partitionable server.

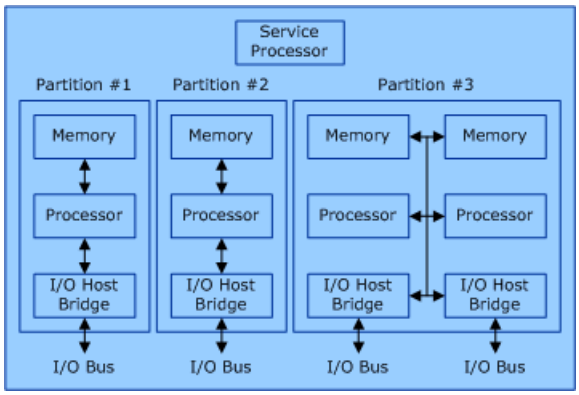
Most existing user-mode applications should continue to run on dynamically partitionable servers without any modification. However, if an application allocates threads for each processor or performs memory allocations that are based on how much physical memory is available, the application can register with the operating system to be notified of changes to the hardware configuration. When the application is notified about a change to the hardware configuration, it can adjust its resource allocation accordingly.

Dynamic Hardware Partitioning Architecture

12/5/2018 • 2 minutes to read • Edit Online

A hardware partitionable server can be configured into one or more isolated hardware partitions. A hardware partition consists of one or more partition units. A partition unit can be a processor, a memory module, or an I/O host bridge.

The following figure shows an example of a hardware partitionable server.



In the previous figure, the server has a total of 12 partition units: four memory modules, four processor modules, and four I/O host bridge modules. Each of these partition units is assigned to one of three hardware partitions. Each hardware partition is completely isolated from the other hardware partitions. The service processor is responsible for the configuration of the hardware partitions. It controls the mapping of the partition units to the hardware partitions and creates isolation between the hardware partitions.

Starting with Windows Server 2008, each partition unit is considered a Plug and Play (PnP) device. Because these devices are PnP, you can add them after the operating system has started.

For more information about how a device driver can register itself with the operating system to receive notification when partition units are dynamically added to the hardware partition, see Driver Notification.

For more information about how an application can register itself with the operating system to receive notification when partition units are dynamically added to the hardware partition, see Application Notification.

Changes to the Number of Processors

6/25/2019 • 4 minutes to read • Edit Online

On a dynamically partitionable server, you can add a processor to a hardware partition at any time. Therefore, you should not make any assumptions about the number of active processors in a hardware partition, the processor affinity value, or the processor number that is assigned to each active processor. The bits that are set in the processor affinity value represent each of the currently active processors in the hardware partition. The particular bits that are set will change if you add a processor to the hardware partition.

If any of the following statements are true for a device driver, you must update the driver so that it will function correctly on a dynamically partitionable server when a processor is dynamically added to a hardware partition:

- The device driver uses the number of active processors in the hardware partition to determine the amount of resources that it uses, such as the amount of memory that it allocates, the number of threads that it creates, or the amount of other resources that it uses. In this situation, the device driver's resource allocation will be incorrect if a processor is dynamically added to the hardware partition. This could adversely affect the performance or behavior of the driver.
- The device driver walks the bits of the processor affinity value. In this situation, the device driver might not work correctly if it cannot handle dynamic changes to the processor affinity value or cannot handle gaps in the sequence of bits that are set.
- The device driver uses the bits in the processor affinity value to assign driver-allocated resources to specific processors. In this situation, the device driver's resource assignments will be incorrect if a processor is dynamically added to the hardware partition. This could adversely affect the performance or behavior of the driver.
- The device driver allocates data structures for each active processor in the hardware partition. In this situation, the device driver could cause adverse behavior, data corruption, or a bug check to occur if it tries to access these data structures for a processor that was dynamically added to the hardware partition.
- The device driver's dispatch routines use the processor number of the processor on which they are running to access data structures or other resources that are assigned to that particular processor. In this situation, the device driver's dispatch routines can cause adverse behavior, data corruption, or a bug check to occur if they try to access these resources for a processor that has been dynamically added to the hardware partition.
- The device driver schedules its interrupt service routines (ISRs), deferred procedure calls (DPCs), or other threads on specific processors. In this situation, the device driver might stop functioning correctly if you add a processor to the hardware partition, and the device driver will be unable to fully use any new processors.
- The device driver does not support resource rebalancing. In this situation, the device driver will be unable to use any new processors that are added to the hardware partition for handling interrupts.
- The device driver uses a load balancing algorithm to distribute the processing of I/O requests across multiple processors. In this situation, the device driver might stop functioning correctly if you add a processor to the hardware partition, and the device driver will be unable to fully use any new processors.

If a device driver is affected by changes to the number of active processors, it must register itself with the operating system to be notified when you add processors to the hardware partition. When the device driver is notified, it can respond as required for safe and optimal operation. For more information about how a device driver can register itself with the operating system, see Driver Notification.

To retrieve the current number of active processors in the hardware partition, device drivers should call the **KeQueryActiveProcessorCount** function. To retrieve the current processor affinity value, device drivers can call either the **KeQueryActiveProcessors** function or the **KeQueryActiveProcessorCount** function.

Note If a device driver allocates data structures for each active processor in the hardware partition and the device driver would fail if the memory allocation for the data structures for a new processor failed, the device driver can allocate enough of these data structures during driver initialization to handle the maximum number of processors that the operating system supports. In this situation, the device driver would not have to allocate new data structures when you add new processors to the hardware partition. However, unless the size of these data structures is fairly small, this can be an inefficient use of memory resources. A device driver can query the maximum number of processors that the operating system supports by calling the **KeQueryMaximumProcessorCount** function.

Important Device drivers should always update any saved value of the number of active processors and the processor affinity when it is notified that you added a processor to the hardware partition.

Important A device driver should not count the number of set bits in the processor affinity value to determine the number of active processors in the hardware partition. We recommended that device drivers call the **KeQueryActiveProcessorCount** function for this purpose. This function returns both the number of active processors and the associated processor affinity value.

Important Device drivers that are built for Windows Vista, Windows Server 2008 and later versions of Windows must not use the **KeNumberProcessors** kernel variable to determine the number of active processors in the hardware partition. The **KeNumberProcessors** kernel variable is obsolete in Windows Vista with Service Pack 1 (SP1), Windows Server 2008, and later versions of Windows.

Changes to the Amount of Physical Memory

12/5/2018 • 2 minutes to read • Edit Online

On a dynamically partitionable server, you can add memory modules to a hardware partition at any time. Therefore, do not make any assumptions about how much physical memory exists in a hardware partition.

If a device driver uses the amount of physical memory in the hardware partition to determine the size of the memory buffers that it allocates, you must update the driver so that it will function correctly on a dynamically partitionable server when you dynamically add memory to the hardware partition.

If a device driver is affected by changes to the amount of physical memory, it must register itself with the operating system to be notified when memory is added to the hardware partition. When the device driver is notified, it can respond as required to ensure safe and optimal operation. For more information about how a device driver can register itself with the operating system, see Driver Notification.

Note Starting with Windows Server 2008, the size of the paged and nonpaged system memory pools do not change after the operating system has started. Therefore, if you add memory to the hardware partition, the amount of memory in these system memory pools does not change.

Hot Replace of Partition Units

12/5/2018 • 2 minutes to read • Edit Online

On a dynamically partitionable server, you can dynamically replace partition units in a hardware partition at any time. This is known as a hot replace operation. When you replace a partition unit, the operating system puts the hardware partition into a pseudo S4 sleep state. To put the hardware partition into this special sleep state, the operating system sends an S4 *set power* power management request to all the device drivers in the system. However, unlike a typical S4 power state, the operating system does not write out the state of the system to a hibernation file.

A device driver must support this pseudo S4 sleep state by correctly handling the *query power* and *set power* power management requests. A device driver should never reject a *query power* request. When a device driver receives an S4 *set power* power management request, it must transition its devices into a D3 device power state and stop all I/O operations. This includes any direct memory access (DMA) transfers that are currently in progress. By putting the driver's devices into a low power state, disabling interrupts, and halting all I/O operations that are in progress, the replace operation can continue without affecting the device driver.

While a device driver's devices are in the D3 power state, the device driver should queue any new I/O requests that it receives. A device driver should use an I/O time-out period for the I/O requests that it processes. This time-out period must be long enough so that the I/O requests will not time out if they are stopped or queued during the replacement of a partition unit. When the operating system resumes from the pseudo S4 sleep state, the device driver can resume processing any stopped or queued I/O requests.

For more information about how to implement support for power management in a device driver, see Power Management.

A device driver must not bind itself to a uniquely identifiable instance of system hardware such as a specific processor. Otherwise, the driver might fail if the partition unit that contains that hardware is replaced in the hardware partition.

Introduction to Driver Notification

6/25/2019 • 5 minutes to read • Edit Online

Starting with Windows Server 2008, the operating system can notify a device driver when a processor or memory module is dynamically added to a hardware partition. There are several different notifications that occur at different stages of the process of a hot add operation. Each of these notifications uses a different notification method to notify the device driver about the event. You can use one or more of these notification methods to have the operating system notify your driver when a hot add operation occurs. Your driver can then respond as required for safe and optimal operation.

The following table identifies the different notification methods and whether they apply to processors, memory, or both processors and memory.

NOTIFICATION METHOD	FOR PROCESSORS	FOR MEMORY
Synchronous driver notification	X	
Asynchronous driver notification	X	X
Memory notification event		X
Resource rebalancing	X	

Synchronous Driver Notification

With Synchronous Driver Notification, the operating system synchronously notifies device drivers that a new processor has been added to the hardware partition. This is the first notification that a device driver receives about a change to the number of processors.

When a new processor is added to the hardware partition, the operating system sends this notification to device drivers after the operating system has started the new processor, but before the operating system begins scheduling threads on the processor. When a device driver receives this notification, it can allocate any per processor data structures and assign any other per processor resources to the new processor. This prepares the device driver to run its dispatch routines, interrupt service routines (ISRs), deferred procedure calls (DPCs), and any other driver threads on the new processor.

A device driver must register itself with the operating system to receive synchronous driver notification. For more information, see Registering for Synchronous Driver Notification.

This notification method is only applicable to processors. There is no synchronous notification mechanism for memory.

Asynchronous Driver Notification

With Asynchronous Driver Notification, the operating system asynchronously notifies device drivers that a new processor or memory module has been added to the hardware partition. Starting with Windows Server 2008, processors and memory modules are considered Plug and Play (PnP) devices. Therefore, the operating system uses the PnP notification mechanism for asynchronous driver notification.

When a new processor or memory module is added to the hardware partition, the operating system sends this notification to device drivers after the operating system has started the new processor or memory device. In the case of a new processor, the operating system does not send this notification to device drivers until after it has started scheduling threads on the new processor.

> **NOTE**
> All PnP notifications are asynchronous. Therefore, these notifications might not be received by a device driver until sometime after the operating system has started the processor or memory module.

When a device driver receives this notification, it can adjust some or all of the following items accordingly:

- Memory buffer and other resource allocations
- Assignment of resources to specific processors
- Scheduling of DPCs and other threads on specific processors
- Load balancing algorithms

> **IMPORTANT**
> When you add a new processor to a hardware partition, the operating system does not send the PnP notification until after the new processor has been started and the operating system has begun scheduling threads on it. If a device driver must perform certain tasks before the operating system begins scheduling threads on the new processor, such as allocating a per processor data structure, you must use the synchronous notification method for the driver.

A device driver must register itself with the operating system to receive asynchronous driver notification. For more information, see Registering for Asynchronous Driver Notification.

Memory Notification Event

With the memory notification event method, you can have your device driver schedule a thread that waits for the operating system to set the **\KernelObjects\HighMemoryCondition** event object. The operating system sets this event object when the amount of free physical memory exceeds a certain value. This event notifies any threads that are waiting on the event object that a significant amount of physical memory is currently available in the system. This event could be an indication that you dynamically added a new memory module to the system. When the operating system sets this event object, your device driver can respond to the event by allocating additional memory buffers.

For more information about the **\KernelObjects\HighMemoryCondition** event object, see Standard Event Objects.

> **IMPORTANT**
> If the operating system sets the **\KernelObjects\HighMemoryCondition** event object, the event only provides an indication that you might have dynamically added a new memory module to the hardware partition. There are other situations that can cause the operating system to set this event object. Therefore, starting with Windows Server 2008, we do not recommend that device drivers use this notification method. Instead, device drivers should use the asynchronous driver notification method.

This method is only applicable to memory. There is no corresponding notification mechanism for processors.

Resource Rebalance

Starting with Windows Server 2008, when you add a new processor to a hardware partition, the operating system initiates a system-wide resource rebalance. Whether a device will participate in such a resource rebalance is determined by the setting of the **DEVPKEY_Device_DHP_Rebalance_Policy** device property for the device. The default behavior for devices in the Network Adapter (Class = Net) device setup class is that they will not participate in resource rebalancing when a new processor is dynamically added to the system. The default behavior for devices in all other device setup classes is that they will participate in resource rebalancing when a new processor is dynamically added to the system.

If a device is a Plug and Play (PnP) device and it participates in such a resource rebalance, the operating system sends **IRP_MN_QUERY_STOP_DEVICE**, **IRP_MN_STOP_DEVICE**, and **IRP_MN_START_DEVICE** PnP IRPs to the driver for the device during the resource rebalancing operation. These PnP requests notify the driver that a hardware change has occurred in the hardware partition. A device driver should support resource rebalancing by correctly handling the **IRP_MN_QUERY_STOP_DEVICE** and **IRP_MN_STOP_DEVICE** PnP requests. A device driver should never reject a **IRP_MN_QUERY_STOP_DEVICE** PnP request.

These PnP requests enable a device driver to fully use the new set of active processors in the hardware partition after you add a new processor. Specifically, a device driver that supports resource rebalancing uses the PnP requests that it receives during the resource rebalance to disconnect its interrupt service routines (ISRs) and reconnect them with the updated processor affinity value. This enables the device driver to use all the currently active processors in the hardware partition, including any new processors, for handling interrupt requests.

Device drivers should queue all I/O requests during resource rebalancing.

For more information about resource rebalancing, see Stopping a Device to Rebalance Resources.

This method is only applicable to processors. The operating system does not initiate a system-wide resource rebalance when you add a new memory module to a hardware partition.

Registering for Synchronous Driver Notification

6/25/2019 • 2 minutes to read • Edit Online

To use synchronous driver notification, a device driver implements a callback function that the operating system calls when you dynamically add a new processor to the hardware partition. The following code example is a prototype for such a callback function:

```
// Prototype for the synchronous
// notification callback function
VOID
  SyncProcessorCallback(
    IN PVOID CallbackContext,
    IN PKE_PROCESSOR_CHANGE_NOTIFY_CONTEXT ChangeContext,
    IN PNTSTATUS OperationStatus
    );
```

A device driver registers for synchronous driver notification by calling the **KeRegisterProcessorChangeCallback** function. A device driver typically calls the **KeRegisterProcessorChangeCallback** function from within its **DriverEntry** function. If the device driver specifies the KE_PROCESSOR_CHANGE_ADD_EXISTING flag, the callback function is immediately called for each active processor that currently exists in the hardware partition, in addition to being called when a new processor is added to the hardware partition. The following code example shows how to register for the synchronous driver notifications:

```
PVOID CallbackRegistrationHandle;
NTSTATUS CallbackStatus = STATUS_SUCCESS;

// The driver's DriverEntry routine
NTSTATUS  DriverEntry(
    PDRIVER_OBJECT DriverObject,
    PUNICODE_STRING RegistryPath
    )
{
  ...

  // Register the callback function
  CallbackRegistrationHandle =
    KeRegisterProcessorChangeCallback(
      SyncProcessorCallback,
      &CallbackStatus,
      KE_PROCESSOR_CHANGE_ADD_EXISTING
      );

  // Check the result
  if (CallbackRegistrationHandle == NULL)
  {
    // Perform any necessary cleanup
    ...

    // Check the callback status
    if (CallbackStatus != STATUS_SUCCESS)
    {
      // Return the error status from the callback function
      return CallbackStatus;
    }
    else
    {
      // Return a generic error status
      return STATUS_UNSUCCESSFUL;
    }
  }

  ...

  return STATUS_SUCCESS;
}
```

When a device driver must stop receiving synchronous driver notifications, such as when it is being unloaded, it must unregister the callback function by calling the **KeDeregisterProcessorChangeCallback** function. A device driver typically calls the **KeDeregisterProcessorChangeCallback** function from within its *Unload* function. The following code example shows how to unregister the callback function:

```
// The driver's Unload routine
VOID
  Unload(
    IN PDRIVER_OBJECT DriverObject
    );
{
  ...

  // Unregister the callback function
  KeDeregisterProcessorChangeCallback(
    CallbackRegistrationHandle
    );

  ...
}
```

Processing a Synchronous Driver Notification

6/25/2019 • 2 minutes to read • Edit Online

When the operating system calls a registered callback function, it passes a pointer to a **KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT** structure in the *ChangeContext* parameter and a pointer to a variable that contains an NTSTATUS code in the *OperationStatus* parameter. The **KE_PROCESSOR_CHANGE_NOTIFY_CONTEXT** structure contains the state of the processor add operation, the processor number of the new processor that is being added, and an NTSTATUS code that is associated with the indicated state.

When a new processor is added to the hardware partition, the operating system calls the registered callback function two times. The operating system first calls the callback function with the **KeProcessorAddStartNotify** state before it starts the new processor. If the operating system successfully adds the new processor, it calls the callback function a second time with the **KeProcessorAddCompleteNotify** state. Otherwise, it calls the callback function a second time with the **KeProcessorAddFailureNotify** state.

If the KE_PROCESSOR_CHANGE_ADD_EXISTING flag was specified when the device driver registered the callback function, the callback function is also called immediately for each active processor that currently exists in the hardware partition. Typically, the callback function will not have to distinguish between when it is called for an existing processor and when it is called for a new processor. For more information about when the operating system calls a registered callback function, see the description of the **KeRegisterProcessorChangeCallback** function.

The following code example shows an implementation of a callback function that processes synchronous driver notifications:

```
// Synchronous notification callback function
VOID
  SyncProcessorCallback(
    IN PVOID CallbackContext,
    IN PKE_PROCESSOR_CHANGE_NOTIFICATION_CONTEXT ChangeContext,
    IN PNTSTATUS OperationStatus
    )
{
  PNTSTATUS CallbackStatus;
  ULONG ProcessorNumber;
  ULONG ProcessorCount;
  KAFFINITY ActiveProcessors;

  // The CallbackContext contains a pointer to a
  // variable that contains an NTSTATUS code. This
  // is used to pass back any error status to the
  // caller of KeRegisterProcessorChangeCallback.
  CallbackStatus =
    (PNTSTATUS)
      CallbackContext;

  // Get the processor number of the new processor
  ProcessorNumber =
    ChangeContext->NtNumber;

  // Switch on the state of the add operation
  switch (ChangeContext->State)
  {
    // Before the operating system has added the new processor
    case KeProcessorAddStartNotify:
```

```
      // Allocate any per-processor data
      // structures for the new processor.
      ...

      // Assign any other per-processor resources
      // to the new processor.
      ...

      // Perform any other necessary preparation
      // for execution of the driver code
      // on the new processor.
      ...

      // If an error occurs such that continuing
      // to add the processor would be fatal
      // (for example, an allocation failure of a
      // per-processor data structure), set the
      // status to an appropriate NTSTATUS code.
      if (...)
      {
        // This returns the status to the operating system.
        *OperationStatus = STATUS_INSUFFICIENT_RESOURCES;

        // This returns the status to the caller of the
        // KeRegisterProcessorChangeCallback function.
        *CallbackStatus = STATUS_INSUFFICIENT_RESOURCES;
      }

      break;

    // The operating system has successfully added the new processor
    case KeProcessorAddCompleteNotify:

      //
      // The following can be performed either here or
      // in an asynchronous notification callback function.
      //

      // Get the current processor count and affinity
      ProcessorCount =
        KeQueryActiveProcessorCount(
          &ActiveProcessors
          );

      // Adjust any resource allocations that are based
      // on the number of processors.
      ...

      // Adjust the assignment of resources that are
      // assigned to specific processors.
      ...

      // Begin scheduling any per-processor threads
      // on the new processor.
      ...

      // Adjust the scheduling of DPCs and other threads
      ...

      // Adjust any load balancing algorithms
      ...

      break;

    // The operating system has failed to add the new processor
    case KeProcessorAddFailureNotify:

      // Clean up and free any per-processor
      // resources that were allocated for the
```

```
        // specified processor during the
        // KeProcessorAddStartNotify state.
        ...

        break;
    }
  }
```

Registering for Asynchronous Driver Notification

6/25/2019 • 2 minutes to read • Edit Online

To use asynchronous driver notification, a device driver implements callback functions that the operating system calls when you dynamically add a processor or memory module to the hardware partition. The following code example shows prototypes for such callback functions:

```
// Prototypes for the asynchronous
// notification callback functions
NTSTATUS
  AsyncProcessorCallback(
    IN PVOID NotificationStructure,
    IN PVOID Context
    );

NTSTATUS
  AsyncMemoryCallback(
    IN PVOID NotificationStructure,
    IN PVOID Context
    );
```

A device driver registers for asynchronous notification by calling the **IoRegisterPlugPlayNotification** function, one time for each of the device driver's callback functions, specifying a pointer to one of the following GUIDs for the *EventCategoryData* parameter:

GUID_DEVICE_PROCESSOR
Register to be notified when a processor is dynamically added to the hardware partition.

GUID_DEVICE_MEMORY
Register to be notified when memory is dynamically added to the hardware partition.

These GUIDs are defined in the header file, Poclass.h.

The following code example shows how to register for both notifications:

```
PVOID ProcessorNotificationEntry;
PVOID MemoryNotificationEntry;
NTSTATUS Status;

Status =
  IoRegisterPlugPlayNotification(
    EventCategoryDeviceInterfaceChange,
    0,
    &GUID_DEVICE_PROCESSOR,
    DriverObject,
    AsyncProcessorCallback,
    NULL,
    &ProcessorNotificationEntry
    );

Status =
  IoRegisterPlugPlayNotification(
    EventCategoryDeviceInterfaceChange,
    0,
    &GUID_DEVICE_MEMORY,
    DriverObject,
    AsyncMemoryCallback,
    NULL,
    &MemoryNotificationEntry
    );
```

Note If a device driver only has to be notified about processors, it does not have to implement a callback function for memory or register for notification about memory. Similarly, if a device driver only has to be notified about memory, it does not have to implement a callback function for processors or register for notification about processors.

When a device driver must stop receiving asynchronous driver notifications, such as when it is being unloaded, it must unregister each callback function by calling the **IoUnregisterPlugPlayNotification** function. The following code example shows how to unregister the callback functions:

```
// Unregister for asynchronous notifications
Status =
  IoUnregisterPlugPlayNotification(
    ProcessorNotificationEntry
    );

Status =
  IoUnregisterPlugPlayNotification(
    MemoryNotificationEntry
    );
```

Processing an Asynchronous Driver Notification

6/25/2019 • 2 minutes to read • Edit Online

When the operating system calls a registered callback function, it passes a pointer to a **DEVICE_INTERFACE_CHANGE_NOTIFICATION** structure in the *NotificationStructure* parameter.

The following code example shows an implementation of a callback function that processes asynchronous driver notifications for processors:

```
// Asynchronous processor notification callback function
NTSTATUS
  AsyncProcessorCallback(
    IN PVOID NotificationStructure,
    IN PVOID Context
    )
{
  PDEVICE_INTERFACE_CHANGE_NOTIFICATION Notification;
  ULONG ProcessorCount;
  KAFFINITY ActiveProcessors;

  Notification =
    (PDEVICE_INTERFACE_CHANGE_NOTIFICATION)
      NotificationStructure;

  // Verify that this notification is about a processor
  // that is being added to the hardware partition.
  if (IsEqualGUID(
        &Notification->Event,
        &GUID_DEVICE_INTERFACE_ARRIVAL
        ))
  {
    // Get the current processor count and affinity
    ProcessorCount =
      KeQueryActiveProcessorCount(
        &ActiveProcessors
        );

    // Adjust any resource allocations that are based
    // on the number of processors.
    ...

    // Adjust the assignment of resources that are
    // assigned to specific processors.
    ...

    // Begin scheduling any per processor threads
    // on the new processor.
    ...

    // Adjust the scheduling of DPCs and other threads
    ...

    // Adjust any load balancing algorithms
    ...
  }

  // Always return success status
 return STATUS_SUCCESS;
}
```

The following code example shows an implementation of a callback function that processes asynchronous driver notifications for memory:

```
// Asynchronous memory notification callback function
NTSTATUS
  AsyncMemoryCallback(
    IN PVOID NotificationStructure,
    IN PVOID Context
    )
{
  PDEVICE_INTERFACE_CHANGE_NOTIFICATION Notification;

  Notification =
    (PDEVICE_INTERFACE_CHANGE_NOTIFICATION)
      NotificationStructure;

  // Verify that this notification is about memory
  // that is being added to the hardware partition.
  if (IsEqualGUID(
        &Notification->Event,
        &GUID_DEVICE_INTERFACE_ARRIVAL
        ))
  {
    // Increase the size of any memory buffers
    // for optimal performance of the driver.
    ...
  }

  // Always return success status
  return STATUS_SUCCESS;
}
```

Introduction to Application Notification

12/21/2018 • 2 minutes to read • Edit Online

Starting with Windows Server 2008, processors and memory modules are considered Plug and Play (PnP) devices. Therefore, the operating system uses the PnP notification mechanism for application notification. The PnP notification mechanism sends WM_DEVICECHANGE window messages to user-mode applications to notify the applications about changes to the hardware in the hardware partition.

When a new processor or memory module is added to the hardware partition, the operating system sends this notification to user-mode applications after the operating system has started the new processor or memory device. In the case of a new processor, the operating system does not send this message to user-mode applications until after it has started scheduling threads on the new processor.

Note All PnP notifications are asynchronous. Therefore, these notifications might not be received by a user-mode application until sometime after the operating system has started the processor or memory module.

When a user-mode application receives this notification, it can adjust some or all of the following items accordingly:

- Per processor memory allocations
- The number of threads in the thread pools of the application
- Memory buffer allocations
- Load balancing algorithms

A user-mode application can get the amount of physical memory that is in the hardware partition by calling the GlobalMemoryStatusEx function. For more information about the **GlobalMemoryStatusEx** function, see the Microsoft Windows SDK documentation.

A user-mode application must register itself with the operating system to receive application notification. For more information, see Registering for Application Notification.

Registering for Application Notification

3/5/2019 • 2 minutes to read • Edit Online

A user-mode application calls the RegisterDeviceNotification function to register itself to be notified when a processor or memory module is dynamically added to the hardware partition. An application calls the **RegisterDeviceNotification** function two times, one time to register for notification of processor events and a second time to register for notification of memory events. The application specifies one of the following GUIDs when it registers for notification of these events:

GUID_DEVICE_PROCESSOR
Registers the application to be notified when a processor is dynamically added to the hardware partition.

GUID_DEVICE_MEMORY
Registers the application to be notified when memory is dynamically added to the hardware partition.

These GUIDs are defined in the header file, Poclass.h.

The following code example shows how to register for both notifications:

```
HWND hWnd;
DEV_BROADCAST_DEVICEINTERFACE ProcessorFilter;
DEV_BROADCAST_DEVICEINTERFACE MemoryFilter;
HDEVNOTIFY ProcessorNotifyHandle;
HDEVNOTIFY MemoryNotifyHandle;

// The following example assumes that hWnd already
// contains a handle to the application window that
// is to receive the WM_DEVICECHANGE messages.

// Initialize the filter for processor event notification
ZeroMemory(
  &ProcessorFilter,
  sizeof(ProcessorFilter)
  );
ProcessorFilter.dbcc_size =
  sizeof(DEV_BROADCAST_DEVICEINTERFACE);
ProcessorFilter.dbcc_devicetype =
  DBT_DEVTYP_DEVICEINTERFACE;
ProcessorFilter.dbcc_classguid =
  GUID_DEVICE_PROCESSOR;

// Register the application window to receive
// WM_DEVICECHANGE messages for processor events.
ProcessorNotifyHandle =
  RegisterDeviceNotification(
    hWnd,
    &ProcessorFilter,
    DEVICE_NOTIFY_WINDOW_HANDLE
    );

// Initialize the filter for memory event notification
ZeroMemory(
  &MemoryFilter,
  sizeof(MemoryFilter)
  );
MemoryFilter.dbcc_size =
  sizeof(DEV_BROADCAST_DEVICEINTERFACE);
MemoryFilter.dbcc_devicetype =
  DBT_DEVTYP_DEVICEINTERFACE;
MemoryFilter.dbcc_classguid =
  GUID_DEVICE_MEMORY;

// Register the application's window to receive
// WM_DEVICECHANGE messages for memory events.
MemoryNotifyHandle =
  RegisterDeviceNotification(
    hWnd,
    &MemoryFilter,
    DEVICE_NOTIFY_WINDOW_HANDLE
    );
```

Note If an application only has to be notified about processors, it does not have to register for notification of memory events. Similarly, if an application only has to be notified about memory, it does not have to register for notification of processor events.

When an application no longer has to receive notification of processor or memory events, it can unregister the window from receiving WM_DEVICECHANGE messages for these events by calling the UnregisterDeviceNotification function. The following code example shows how to unregister for the application notifications:

```
// Unregister the application window from receiving
// WM_DEVICECHANGE messages for processor events.
UnregisterDeviceNotification(ProcessorNotifyHandle);

// Unregister the application window from receiving
// WM_DEVICECHANGE messages for memory events.
UnregisterDeviceNotification(MemoryNotifyHandle);
```

For more information about the **RegisterDeviceNotification** and **UnregisterDeviceNotification** functions, see the Microsoft Windows SDK documentation.

Processing an Application Notification

3/5/2019 • 3 minutes to read • Edit Online

How a user-mode application processes WM_DEVICECHANGE messages depends on whether the application is based purely on the Win32 API or whether it is based on the Microsoft Foundation Class (MFC) library.

Win32 applications

Win32-based applications process the messages that are sent to the application's window(s) by implementing a *Window Procedure*. For more information about window procedures, see the Window Procedures topic in the Microsoft Windows SDK documentation.

The following code example shows how to process WM_DEVICECHANGE messages in a Win32-based application:

```
// Prototype for the function that handles the
// processing of WM_DEVICECHANGE messages.
LRESULT
OnDeviceChange(
  WPARAM wParam,
  LPARAM lParam
  );

// The application's message processing function
// for the window that receives the WM_DEVICECHANGE
// messages.
LRESULT CALLBACK
WindowProc(
  HWND hWnd,
  UINT uMsg,
  WPARAM wParam,
  LPARAM lParam
  )
{
  switch (uMsg)
  {
      .
      .  // Cases for other messages
      .
    // Device change message
    case WM_DEVICECHANGE:
      OnDeviceChange(wParam, lParam);
      break;
      .
      .  // Cases for other messages
      .
    // Catchall for all messages that are
    // not handled by the application.
    default:
      return DefWindowProc(
               hWnd,
               uMsg,
               wParam,
               lParam
               );
  }

  return 0;
}

// The function that handles the processing
```

```
// of WM_DEVICECHANGE messages.
LRESULT
OnDeviceChange(
  WPARAM wParam,
  LPARAM lParam
  )
{
  PDEV_BROADCAST_HDR devHdr;
  PDEV_BROADCAST_DEVICEINTERFACE devInterface;
  HANDLE ProcessHandle;
  DWORD_PTR ProcessAffinityMask;
  DWORD_PTR SystemAffinityMask;
  DWORD_PTR ChangedAffinityMask;
  MEMORYSTATUSEX MemoryStatus;

  // Check whether the message is a device arrival message
  if (wParam == DBT_DEVICEARRIVAL)
  {
    // Get a pointer to the structure header
    devHdr = (PDEV_BROADCAST_HDR)lParam;

    // Check whether the message is about a device interface
    if (devHdr->dbch_devicetype == DBT_DEVTYP_DEVICEINTERFACE)
    {
      // Get a pointer to the device interface structure
      devInterface = (PDEV_BROADCAST_INTERFACE)devHdr;

      // Check whether this is a message about a processor
      if (IsEqualGUID(
            devInterface->dbcc_classguid,
            GUID_DEVICE_PROCESSOR
            ))
      {
        // Get a handle to the current process
        ProcessHandle =
          GetCurrentProcess();

        // Get the current process and system affinity masks
        GetProcessAffinityMask(
          ProcessHandle,
          &ProcessAffinityMask,
          &SystemAffinityMask
          );

        // Get a mask of any change to the set of processors
        ChangedAffinityMask =
          ProcessAffinityMask ^ SystemAffinityMask;

        // Perform any per processor memory allocation
        // for any new processors
        ...

        // Set the process affinity mask to use all the
        // active processors in the hardware partition.
        SetProcessAffinityMask(
          ProcessHandle,
          SystemAffinityMask
          );

        // Adjust the number of threads in any thread
        // pools as needed for optimal performance.
        ...
      }

      // Check whether this is a message about memory
      else if (IsEqualGUID(
                  devInterface->dbcc_classguid,
                  GUID_DEVICE_MEMORY
                  ))
```

```
      {
        // Get the current memory status
        GlobalMemoryStatusEx(&MemoryStatus);

        // Note: MemoryStatus.ullTotalPhys contains
        // the amount of physical memory in the
        // hardware partition.

        // Adjust the memory buffer allocations
        // as needed for optimal performance.
        ...
      }
    }
  }

  return 0;
}
```

MFC applications

The MFC framework processes the messages that are sent to an MFC-based application's window(s). An MFC-based application must implement an OnDeviceChange member function for the application's window class that receives the WM_DEVICECHANGE messages.

The following code example shows how to implement an **OnDeviceChange** member function in an MFC-based application:

```
afx_msg BOOL
CAppWnd::OnDeviceChange(
  UINT nEventType,
  DWORD_PTR dwData
  )
{
  PDEV_BROADCAST_HDR devHdr;
  PDEV_BROADCAST_DEVICEINTERFACE devInterface;
  HANDLE ProcessHandle;
  DWORD_PTR ProcessAffinityMask;
  DWORD_PTR SystemAffinityMask;
  DWORD_PTR ChangedAffinityMask;
  MEMORYSTATUSEX MemoryStatus;

  if (nEventType == DBT_DEVICEARRIVAL)
  {
    devHdr = (PDEV_BROADCAST_HDR)dwData;

    if (devHdr->dbch_devicetype == DBT_DEVTYP_DEVICEINTERFACE)
    {
      devInterface = (PDEV_BROADCAST_INTERFACE)devHdr;

      if (IsEqualGUID(
            devInterface->dbcc_classguid,
            GUID_DEVICE_PROCESSOR
            ))
      {
        // Get a handle to the current process
        ProcessHandle =
          GetCurrentProcess();

        // Get the current process and system affinity masks
        GetProcessAffinityMask(
          ProcessHandle,
          &ProcessAffinityMask,
          &SystemAffinityMask
          );

        // Get a mask of any change to the set of processors
        ChangedAffinityMask
```

```
        ChangedAffinityMask =
          ProcessAffinityMask ^ SystemAffinityMask;

        // Perform any per processor memory allocation
        // for the new processors
        ...

        // Set the process affinity mask to use all the
        // active processors in the hardware partition.
        SetProcessAffinityMask(
          ProcessHandle,
          SystemAffinityMask
          );

        // Adjust the number of threads in any thread
        // pools as needed for optimal performance.
        ...
      }
      else if (IsEqualGUID(
                devInterface->dbcc_classguid,
                GUID_DEVICE_MEMORY
                ))
      {
        // Get the current memory status
        GlobalMemoryStatusEx(&MemoryStatus);

        // Note: MemoryStatus.ullTotalPhys contains
        // the amount of physical memory in the
        // hardware partition.

        // Adjust the memory buffer allocations
        // as needed for optimal performance.
        ...
      }
    }
  }

  return TRUE;
}
```

Using NTSTATUS Values

6/25/2019 • 2 minutes to read • Edit Online

Many kernel-mode standard driver routines and driver support routines use the NTSTATUS type for return values. Additionally, drivers provide an NTSTATUS-typed value in an IRP's **IO_STATUS_BLOCK** structure when completing IRPs. The NTSTATUS type is defined in Ntdef.h, and system-supplied status codes are defined in Ntstatus.h. (Vendors can also define private status codes, although they rarely need to. For more information, see Defining New NTSTATUS Values.)

NTSTATUS values are divided into four types: success values, informational values, warnings, and error values.

Numerous values are assigned to each type. A common mistake, when testing for a successful return from a routine, is to compare the routine's return value with STATUS_SUCCESS. This comparison checks for only one of several success values.

When testing a return value, you should use one of the following system-supplied macros (defined in Ntdef.h):

NT_SUCCESS(*Status*)
Evaluates to **TRUE** if the return value specified by *Status* is a success type (0 – 0x3FFFFFFF) or an informational type (0x40000000 – 0x7FFFFFFF).

NT_INFORMATION(*Status*)
Evaluates to **TRUE** if the return value specified by *Status* is an informational type (0x40000000 – 0x7FFFFFFF).

NT_WARNING(*Status*)
Evaluates to **TRUE** if the return value specified by *Status* is a warning type (0x80000000 – 0xBFFFFFFF).

NT_ERROR(*Status*)
Evaluates to **TRUE** if the return value specified by *Status* is an error type (0xC0000000 - 0xFFFFFFFF).

For example, suppose a driver calls **IoRegisterDeviceInterface** to register a device interface. If the driver checks the return value using the NT_SUCCESS macro, the macro will evaluate to **TRUE** if the routine returns STATUS_SUCCESS, which indicates no errors, or if it returns the informational status STATUS_OBJECT_NAME_EXISTS, which indicates that the device interface is already registered.

As another example, suppose a driver calls **ZwEnumerateKey** to enumerate the subkeys of a specified registry key. If the NT_SUCCESS macro evaluates to **FALSE**, it might be because the routine returned STATUS_INVALID_PARAMETER, which is an error code, or because the routine returned STATUS_NO_MORE_ENTRIES, which is a warning code.

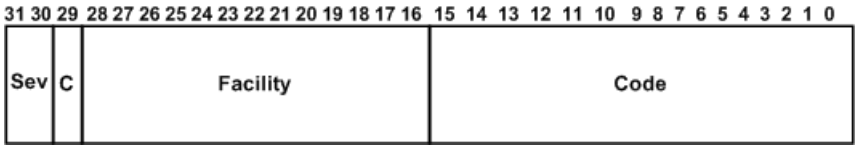
As a final example, suppose a driver sends an IRP that causes a lower-level driver to read information from a device. If the requesting driver specifies a buffer that is too small to receive any information, the lower-level driver might respond by returning STATUS_BUFFER_TOO_SMALL, which is an error code. If the first driver specifies a buffer that can receive some, but not all, of the requested information, the lower-level driver might respond by supplying as much data as possible and then returning STATUS_BUFFER_OVERFLOW, which is a warning code. Note that if the first driver tests the status value using NT_SUCCESS or NT_ERROR incorrectly, it might inadvertently drop some of the information received.

Defining New NTSTATUS Values

6/25/2019 • 2 minutes to read • Edit Online

Drivers can define custom IO_ERR_*XXX* constants to use as **ErrorCode** values when logging errors. Pairs of drivers that are written together can also define custom STATUS_*XXX* values for **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests.

The following diagram shows the bit fields in a 32-bit NTSTATUS value.



The **Sev** field shown in the preceding diagram indicates the severity code, which must be one of the following system-defined values:

STATUS_SEVERITY_SUCCESS
Indicates a successful NTSTATUS value, such as STATUS_SUCCESS, or the value IO_ERR_RETRY_SUCCEEDED in error log packets.

STATUS_SEVERITY_INFORMATIONAL
Indicates an informational NTSTATUS value, such as STATUS_SERIAL_MORE_WRITES.

STATUS_SEVERITY_WARNING
Indicates a warning NTSTATUS value, such as STATUS_DEVICE_PAPER_EMPTY.

STATUS_SEVERITY_ERROR
Indicates an error NTSTATUS value, such as STATUS_INSUFFICIENT_RESOURCES for a **FinalStatus** value or IO_ERR_CONFIGURATION_ERROR for an **ErrorCode** value in error log packets.

Most public IO_ERR_*XXX* constants belong to the STATUS_SEVERITY_ERROR category.

The **Facility** code specifies the facility that generated the error. For new IO_ERR_*XXX* values, drivers specify the FACILITY_IO_ERROR_CODE value for **Facility**. For custom STATUS_*XXX* values, the meaning of different values for **Facility** is driver-defined.

The **C** bit specifies if the value is customer-defined or Microsoft-defined. The bit is set for customer-defined values and clear for Microsoft-defined values.

Drivers can define new IO_ERR_*XXX* values to identify custom error messages in the system event log. For a description of how to define the NTSTATUS values and the error messages that they identify, see Defining Custom Error Types.

Pairs of drivers can define driver-specific STATUS_*XXX* values to communicate information about privately defined **IRP_MJ_INTERNAL_DEVICE_CONTROL** requests from the lower to the higher driver of the pair.

The class driver must map any private STATUS_*XXX* value to a system-defined NTSTATUS value when it completes an IRP if an existing higher-level driver's *IoCompletion* routine might be called for that IRP.

For paired display and video miniport drivers, the video port driver does the mapping between public STATUS_*XXX* values and the Win32-defined constants returned by video miniport drivers. For more information, see Video Miniport Drivers in the Windows 2000 Display Driver Model.

Drivers cannot use custom NTSTATUS values for IRPs that can be received in user mode, because only the system-

defined values can be translated into Win32 error codes.

Singly and Doubly Linked Lists

6/25/2019 • 9 minutes to read • Edit Online

Singly Linked Lists

The operating system provides built-in support for singly linked lists that use **SINGLE_LIST_ENTRY** structures. A singly linked list consists of a list head plus some number of list entries. (The number of list entries is zero if the list is empty.) Each list entry is represented as a **SINGLE_LIST_ENTRY** structure. The list head is also represented as a **SINGLE_LIST_ENTRY** structure.

Each **SINGLE_LIST_ENTRY** structure contains a **Next** member that points to another **SINGLE_LIST_ENTRY** structure. In the **SINGLE_LIST_ENTRY** structure that represents the list head, the **Next** member points to the first entry in the list, or is NULL if the list is empty. In the **SINGLE_LIST_ENTRY** structure that represents an entry in the list, the **Next** member points to the next entry of the list, or is NULL if this entry is the last in the list.

The routines that manipulate a singly linked list take a pointer to a **SINGLE_LIST_ENTRY** that represents the list head. They update the **Next** pointer so that it points to the first entry of the list after the operation.

Suppose that the *ListHead* variable is a pointer to the **SINGLE_LIST_ENTRY** structure that represents the list head. A driver manipulates *ListHead* as follows:

- To initialize the list as empty, set *ListHead*->**Next** to be **NULL**.
- To add a new entry to the list, allocate a **SINGLE_LIST_ENTRY** to represent the new entry, and then call **PushEntryList** to add the entry to beginning of the list.
- Pop the first entry off the list by using **PopEntryList**.

A **SINGLE_LIST_ENTRY**, by itself, only has a **Next** member. To store your own data in the lists, embed the **SINGLE_LIST_ENTRY** as a member of the structure that describes the list entry, as follows.

```
typedef struct {
  // driver-defined members
  .
  .
  .
  SINGLE_LIST_ENTRY SingleListEntry;

  // other driver-defined members
  .
  .
  .
} XXX_ENTRY;
```

To add a new entry to the list, allocate an **XXX_ENTRY** structure, and then pass a pointer to the **SingleListEntry** member to **PushEntryList**. To convert a pointer to the **SINGLE_LIST_ENTRY** back to an **XXX_ENTRY**, use **CONTAINING_RECORD**. Here is an example of routines that insert and remove driver-defined entries from a singly linked list.

```
typedef struct {
  PVOID DriverData1;
  SINGLE_LIST_ENTRY SingleListEntry;
  ULONG DriverData2;
} XXX_ENTRY, *PXXX_ENTRY;

void
PushXxxEntry(PSINGLE_LIST_ENTRY ListHead, PXXX_ENTRY Entry)
{
    PushEntryList(ListHead, &(Entry->SingleListEntry));
}

PXXX_ENTRY
PopXxxEntry(PSINGLE_LIST_ENTRY ListHead)
{
    PSINGLE_LIST_ENTRY SingleListEntry;
    SingleListEntry = PopEntryList(ListHead);
    return CONTAINING_RECORD(SingleListEntry, XXX_ENTRY, SingleListEntry);
}
```

The system also provides atomic versions of the list operations, **ExInterlockedPopEntryList** and **ExInterlockedPushEntryList**. Each takes an additional spin lock parameter. The routine acquires the spin lock before it updates the list, and then the routine releases the spin lock after the operation is completed. While the lock is held, interrupts are disabled. Each operation on the list must use the same spin lock to ensure that each such operation on the list is synchronized with every other operation. You must use the spin lock only with these **ExInterlocked*Xxx*List** routines. Do not use the spin lock for any other purpose. Drivers can use the same lock for multiple lists, but this behavior increases lock contention so drivers should avoid it.

For example, the **ExInterlockedPopEntryList** and **ExInterlockedPushEntryList** routines can support sharing of a singly linked list by a driver thread running at IRQL = PASSIVE_LEVEL and an ISR running at DIRQL. These routines disable interrupts when the spin lock is held. Thus, the ISR and driver thread can safely use the same spin lock in their calls to these **ExInterlocked*Xxx*List** routines without risking a deadlock.

Do not mix calls to the atomic and non-atomic versions of the list operations on the same list. If the atomic and non-atomic versions are run simultaneously on the same list, the data structure might become corrupted and the computer might stop responding or bug check (that is, *crash*). (You cannot acquire the spin lock while calling the non-atomic routine as an alternative to mixing calls to atomic and non-atomic versions of list operations. Using the spin lock in this fashion is not supported and might still cause list corruption.)

The system also provides an alternative implementation of atomic singly linked lists that is more efficient. For more information, see Sequenced Singly Linked Lists.

Doubly Linked Lists

The operating system provides built-in support for doubly linked lists that use **LIST_ENTRY** structures. A doubly linked list consists of a list head plus some number of list entries. (The number of list entries is zero if the list is empty.) Each list entry is represented as a **LIST_ENTRY** structure. The list head is also represented as a **LIST_ENTRY** structure.

Each **LIST_ENTRY** structure contains an **Flink** member and a **Blink** member. Both members are pointers to **LIST_ENTRY** structures.

In the **LIST_ENTRY** structure that represents the list head, the **Flink** member points to the first entry in the list and the **Blink** member points to the last entry in the list. If the list is empty, then **Flink** and **Blink** of the list head point to the list head itself.

In the **LIST_ENTRY** structure that represents an entry in the list, the **Flink** member points to the next entry in the list, and the **Blink** member points to the previous entry in the list. (If the entry is the last one in the list, **Flink** points to the list head. Similarly, if the entry is the first one in the list, **Blink** points to the list head.)

(While these rules may seem surprising at first glance, they allow the list insertion and removal operations to implemented with no conditional code branches.)

The routines that manipulate a doubly linked list take a pointer to a **LIST_ENTRY** that represents the list head. These routines update the **Flink** and **Blink** members in the list head so that these members point to the first and last entries in the resulting list.

Suppose that the *ListHead* variable is a pointer to the **LIST_ENTRY** structure that represents the list head. A driver manipulates *ListHead* as follows:

- To initialize the list as empty, use **InitializeListHead**, which initializes *ListHead*->**Flink** and *ListHead*->**Blink** to point to *ListHead*.
- To insert a new entry at the head of the list, allocate a **LIST_ENTRY** to represent the new entry, and then call **InsertHeadList** to insert the entry at the beginning of the list.
- To append a new entry to the tail of the list, allocate a **LIST_ENTRY** to represent the new entry, and then call **InsertTailList** to insert the entry at the end of the list.
- To remove the first entry from the list, use **RemoveHeadList**. This returns a pointer to the removed entry from the list, or to *ListHead* if the list is empty.
- To remove the last entry from the list, use **RemoveTailList**. This returns a pointer to the removed entry from the list, or to *ListHead* if the list is empty.
- To remove a specified entry from the list, use **RemoveEntryList**.
- To check to see if a list is empty, use **IsListEmpty**.
- To append a list to the tail of another list, use **AppendTailList**.

A **LIST_ENTRY**, by itself, only has **Blink** and **Flink** members. To store your own data in the lists, embed the **LIST_ENTRY** as a member of the structure that describes the list entry, as follows.

```
typedef struct {
  // driver-defined members
  .
  .
  .
  LIST_ENTRY ListEntry;

  // other driver-defined members.
  .
  .
  .
} XXX_ENTRY;
```

To add a new entry to a list, allocate an **XXX_ENTRY** structure, and then pass a pointer to the **ListEntry** member to **InsertHeadList** or **InsertTailList**. To convert a pointer to a **LIST_ENTRY** back to an **XXX_ENTRY**, use **CONTAINING_RECORD**. For an example of this technique, using singly linked lists, see Singly Linked Lists above.

The system also provides atomic versions of the list operations, **ExInterlockedInsertHeadList**, **ExInterlockedInsertTailList**, and **ExInterlockedRemoveHeadList**. (Note that there is no atomic version of **RemoveTailList** or **RemoveEntryList**.) Each routine takes an additional spin lock parameter. The routine acquires the spin lock before updating the list and then releases the spin lock after the operation is completed. While the lock is held, interrupts are disabled. Each operation on the list must use the same spin lock to ensure that each such operation on the list is synchronized with every other. You must use the spin lock only with these **ExInterlocked*Xxx*List** routines. Do not use the spin lock for any other purpose. Drivers can use the same lock for

multiple lists, but this behavior increases lock contention so drivers should avoid it.

For example, the **ExInterlockedInsertHeadList**, **ExInterlockedInsertTailList**, and **ExInterlockedRemoveHeadList** routines can support sharing of a doubly linked list by a driver thread running at IRQL = PASSIVE_LEVEL and an ISR running at DIRQL. These routines disable interrupts when the spin lock is held. Thus, the ISR and driver thread can safely use the same spin lock in their calls to these **ExInterlocked*Xxx*List** routines without risking a deadlock.

Do not mix calls to the atomic and non-atomic versions of the list operations on the same list. If the atomic and non-atomic versions are run simultaneously on the same list, the data structure might become corrupt and the computer might stop responding or bug check (that is, *crash*). (You cannot work acquire the spin lock while calling the non-atomic routine to avoid mixing calls to atomic and non-atomic versions of the list operations. Using the spin lock in this fashion is not supported and might still cause list corruption.)

Sequenced Singly Linked Lists

A sequenced singly linked list is an implementation of singly linked lists that supports atomic operations. It is more efficient for atomic operations than the implementation of singly linked lists described in Singly Linked Lists.

An **SLIST_HEADER** structure is used to describe the head of a sequenced singly linked list, while **SLIST_ENTRY** is used to describe an entry in the list.

A driver manipulates the list as follows:

- To initialize an **SLIST_HEADER** structure, use **ExInitializeSListHead**.
- To add a new entry to the list, allocate a **SLIST_ENTRY** to represent the new entry, and then call **ExInterlockedPushEntrySList** to add the entry to the beginning of the list.
- Pop the first entry off the list by using **ExInterlockedPopEntrySList**.
- To clear the list completely, use **ExInterlockedFlushSList**.
- To determine the number of entries in the list, use **ExQueryDepthSList**.

A **SLIST_ENTRY**, by itself, only has a **Next** member. To store your own data in the lists, embed the **SLIST_ENTRY** as a member of the structure that describes the list entry, as follows.

```
typedef struct
{
  // driver-defined members
  .
  .
  .
  SLIST_ENTRY SListEntry;
  // other driver-defined members
  .
  .
  .

} XXX_ENTRY;
```

To add a new entry to the list, allocate an **XXX_ENTRY** structure, and then pass a pointer to the **SListEntry** member to **ExInterlockedPushEntrySList**. To convert a pointer to the **SLIST_ENTRY** back to an **XXX_ENTRY**, use **CONTAINING_RECORD**. For an example of this technique, using non-sequenced singly linked lists, see Singly Linked Lists.

Warning For 64-bit Microsoft Windows operating systems, **SLIST_ENTRY** structures must be 16-byte aligned.

Windows XP and later versions of Windows provide optimized versions of the sequenced singly linked list functions that are not available in Windows 2000. If your driver uses these functions and also must run with

Windows 2000, the driver must define the _WIN2K_COMPAT_SLIST_USAGE flag, as follows:

```
#define _WIN2K_COMPAT_SLIST_USAGE
```

For x86-based processors, this flag causes the compiler to use versions of the sequenced singly linked list functions that are compatible with Windows 2000.

Handling Exceptions

6/25/2019 • 2 minutes to read • Edit Online

The operating system uses structured exception handling to signal certain kinds of errors. A routine called by a driver can raise an exception that the driver must handle.

The system traps the following general kinds of exceptions:

1. Hardware-defined faults or traps, such as,
   - Access violations (see below)
   - Data-type misalignments (such as a 16-bit entity aligned on an odd-byte boundary)
   - Illegal and privileged instructions
   - Invalid lock sequences (attempting to execute an invalid sequence of instructions within an interlocked section of code)
   - Integer divides by zero and overflows
   - Floating-point divides by zero, overflows, underflows, and reserved operands
   - Breakpoints and single step execution (to support debuggers)
2. System software-defined exceptions, such as,
   - Guard-page violations (attempting to load or store data from or to a location within a guard page)
   - Page-read errors (attempting to read a page into memory and encountering a concurrent I/O error)

An *access violation* is an attempt to perform an operation on a page that is not permitted under the current page protection settings. Access violations occur in the following situations:

- An invalid read or write operation, such as writing to a read-only page.
- To access memory beyond the limit of the current program's address space (known as a length violation).
- To access a page that is currently resident but dedicated to the use of a system component. For example, user-mode code is not allowed access a page that the kernel is using.

If an operation might cause an exception, the driver should enclose the operation in a **try/except** block. Accesses of locations in user-mode are typical causes of exceptions. For example, the **ProbeForWrite** routine checks that the driver can actually write to a user-mode buffer. If it cannot, the routine raises a STATUS_ACCESS_VIOLATION exception. In the following code example, the driver calls **ProbeForWrite** in a **try/except** so that it can handle the resulting exception, if one should occur.

```
try {
    ...
    ProbeForWrite(Buffer, BufferSize, BufferAlignment);

    /* Note that any access (not just the probe, which must come first,
     * by the way) to Buffer must also be within a try-except.
     */
    ...
} except (EXCEPTION_EXECUTE_HANDLER) {
    /* Error handling code */
    ...
}
```

Drivers must handle any raised exceptions. An exception that is not handled causes the system to bug check. The

driver that causes the exception to be raised must handle it: a lower-level driver cannot rely on a higher-level driver to handle the exception.

Drivers can directly raise an exception, by using the **ExRaiseAccessViolation**, **ExRaiseDatatypeMisalignment**, or **ExRaiseStatus** routines. The driver must handle any exceptions that these routines raise.

The following is a partial list of routines that, at least in certain situations, can raise an exception:

- **MmMapLockedPages**
- **MmProbeAndLockPages**
- **ProbeForRead**
- **ProbeForWrite**

Memory accesses to user-mode buffers can also cause access violations. For more information, see Errors in Referencing User-Space Addresses.

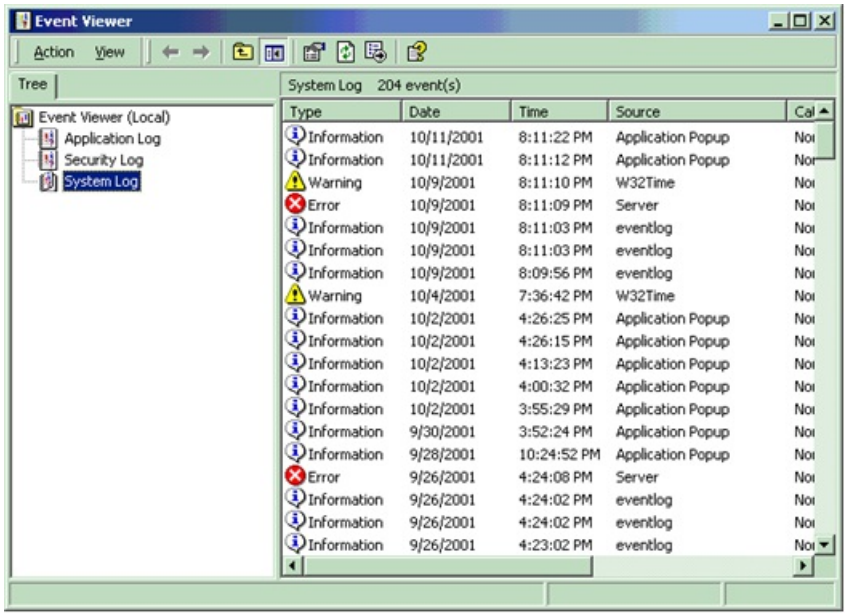
Note that structured exception handling is distinct from C++ exceptions. The kernel does not support C++ exceptions.

For more information about structured exception handling, see the Microsoft Windows SDK, and the Visual Studio documentation.

Logging Errors

12/5/2018 • 2 minutes to read • Edit Online

Drivers, like most Microsoft Windows system components, can log errors to the system event log. The errors are visible in the Event Viewer.



This section includes the following topics:

Writing to the System Event Log

Defining Custom Error Types

Registering as a Source of Error Messages

Writing to the System Event Log

6/25/2019 • 2 minutes to read • Edit Online

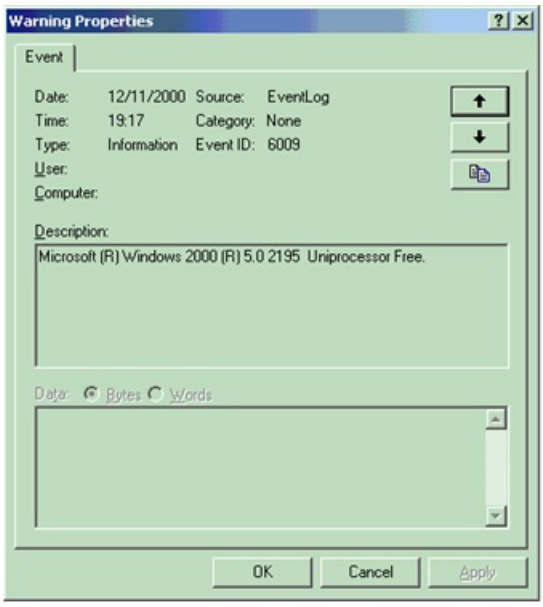
Errors are specified by their NTSTATUS value. The system predefines particular NTSTATUS values that can be used by drivers, and driver writers can define additional errors. Note that only certain NTSTATUS values can be used when logging errors.

Each NTSTATUS value that can be used when logging errors has an associated error message. For example, the parallel port driver uses the NTSTATUS value PAR_INTERRUPT_CONFLICT to represent hardware interrupt conflicts, with message text "Interrupt conflict detected for %1".

The Event Viewer displays the message text in the **Description** text box on the log entry's property sheet. If the message text string contains "%1", the Event Viewer replaces it with the name of the device that logged the entry. The message text can contain additional parameters of the form "%2", "%3", and so on. When the driver logs the error, it can provide string values for those parameters. These string values are known as *insertion strings*. The Event Viewer will automatically insert them in place of the percent values.

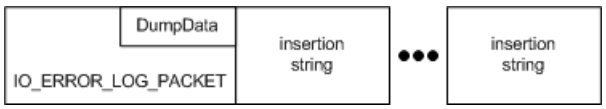
The driver can also include binary data in the log entry, known as *dump data*. The Event Viewer displays the dump data in the **Data** text box of the log entry's property sheet.

You can bring up the property sheet for a log entry by double-clicking the entry in the Event Viewer. The following screen shot shows a sample log entry property sheet.



Drivers use the **IoAllocateErrorLogEntry** routine to allocate an error log entry. Log entries consist of a variable-length **IO_ERROR_LOG_PACKET** header, followed by insertion strings.

The following diagram shows the layout of an error log entry in memory.



The **ErrorCode** member of **IO_ERROR_LOG_PACKET** specifies the NTSTATUS value of the error. The **DumpData** member specifies any dump data for the log entry. **DumpData** is a variable-sized array, whose size is specified by the **DumpDataSize** member. Drivers specify the beginning of the first insertion string with the **StringOffset** member, and the number of strings in the **NumberOfStrings** member. Each insertion string itself is a null-terminated Unicode string.

Once the driver fills out the allocated error log entry, it writes the entry to the error log by using **IoWriteErrorLogEntry**. **IoWriteErrorLogEntry** automatically frees the memory allocated for the log entry. Drivers can use **IoFreeErrorLogEntry** to free any unused log entries.

Predefined error codes (of the form IO_ERR_*XXX*) are defined in the ntiologc.h header file that is included with the Windows Driver Kit (WDK). The error message associated with each error code can be found in the comments for ntiologc.h, next to the error code's declaration. To use a predefined error code, the driver must register the system file, iologmsg.dll, as the source of the associated error messages. For further information, see Registering as a Source of Error Messages.

Drivers can also define their own custom error types, and associated error messages. For further information, see Defining Custom Error Types.

Defining Custom Error Types

12/21/2018 • 5 minutes to read • Edit Online

Drivers can specify their own error types and error messages. To define a custom error message, you must first define a new IO_ERR_*XXX* value to specify as the **ErrorCode** member of the error log entry. The Event Viewer uses the IO_ERR_*XXX* value to look up the driver's error message.

To support custom error messages in your driver, follow these steps:

1. Create a message text file that specifies the custom IO_ERR_*XXX* value and the corresponding error messages. For further information, see Creating the Error Message Text File.
2. Compile the error message text file to a resource, and attach the resource to the driver image. For further information, see Compiling the Error Message Text File.
3. Register the driver image as containing error messages. For further information, see Registering as a Source of Error Messages.

Creating the Error Message Text File

The definition of a driver's custom IO_ERR_*XXX* values and matching error message templates are attached as a message table resource to the driver image. You can describe the messages for a driver in a message text file (which has an .mc file name extension).

A message text file consists of two sections: a header section and a message section. The header section permits the declaration of symbolic names for numerical values, while the message section specifies the IO_ERR_*XXX* values and matching error message templates.

For an example of a message text file, see the Serlog.mc file in the Serial driver sample available on GitHub.

Header Section

The header section must contain this line:

```
MessageIdTypedef=NTSTATUS
```

This ensures that the type of IO_ERR_*XXX* values generated by the Message Compiler is declared to be NTSTATUS.

The other directives that appear in the header section define symbolic values that are used in place of numeric values in the message section.

The **SeverityNames** and **FacilityNames** directives define symbolic values for the severity and facility fields of NTSTATUS values. The directives are of the form *keyword***= (** *values* **)**, where *values* consists of one or more statements of the form *name* **=** *value* **:** *header_name*, separated by white space. The *name* parameter is the name you use the specify the numeric *value* in the message text file, while the *header_name* is the name for this value declared in the C header file generated by the Message Compiler. The **:** *header_name* clause is optional.

Here is an example of a header declaration of symbolic names for severity codes:

```
SeverityNames = (
  Success       = 0x0:STATUS_SEVERITY_SUCCESS
  Informational = 0x1:STATUS_SEVERITY_INFORMATIONAL
  Warning       = 0x2:STATUS_SEVERITY_WARNING
  Error         = 0x3:STATUS_SEVERITY_ERROR
)
```

The **LanguageNames** directive defines symbolic values for locale IDs (LCID). The directive is of the form **LanguageNames = (** *values* **)**, where *values* consists of one or more statements of the form *language_name* **=** *lcid* **:** *langfile*, separated by white space. The *language_name* parameter is the name you use in place of *lcid* in the message text file, while the *filename* specifies a unique file name (without extension). When the Message Compiler generates the resource script from the message text file, it stores all of the string resources for this language in a file named *langfile.bin*.

Message Section

Each message definition begins with the definition of the custom IO_ERR_*XXX* value that the driver uses to report this particular type of error. The IO_ERR_*XXX* value is defined by a series of *keyword* = *value* pairs. The possible keywords and their meaning are as follows.

KEYWORD	VALUE
MessageId	Code field of the new IO_ERR_*XXX* value.
Severity	Severity field of the new IO_ERR_*XXX* value. The specified value must be one of the symbolic names defined by the **SeverityNames** header directive.
Facility	Facility field of the new IO_ERR_*XXX* value. The specified value must be one of the symbolic names defined by the **FacilityNames** header directive.
SymbolicName	The symbolic name for the new IO_ERR_*XXX* value. The Message Compiler generates a C header file that contains a **#define** declaration of the name as the corresponding NTSTATUS value. The driver uses that name when specifying the error type.

The first keyword must always be **MessageId**.

The rest of the message definition consists of one or more localized versions of the error message. Each version is of the form:

```
Language=language_name
localized_message
```

The *language_name* value, which must be one of the symbolic names defined by the **LanguageNames** header directive, specifies the language of the message text. The localized message text itself consists of a Unicode string. Any embedded strings of the form "%*n*" will be treated as templates that the Event Viewer will replace when the error is logged. The "%1" string is replaced with the name of the driver's device object, while "%2" through "%*n*" are replaced with any insertion strings provided by the driver.

The message definition is terminated by a single period alone on a line.

If you define custom error messages, you should not use insertion strings unless necessary. Insertion strings cannot be localized, so they should be used for strings that are language-independent, such as numbers or file names. Most drivers do not use insertion strings.

Compiling the Error Message Text File

Use the Message Compiler (mc.exe) to compile your message text file into a resource script file (which has an .rc file name extension). A command of the form

```
mc filename.mc
```

causes the Message Compiler to generate the following files:

- *filename*.h, a header file that contains declarations of each custom IO_ERR_*XXX* value in *filename.mc*.
- *filename*.rc, a resource script.
- One file for each language that appears in the message text file. Each of these files stores all of the error message string resources for one language. The file for each language is named *langfile.bin*, where *langfile* is the value specified for the language in the message text file's **LanguageNames** directive.

More information about the Message Compiler can be found in the Microsoft Windows SDK.

The Resource Compiler converts a resource script to a resource file that you can attach to your driver image. If you use the Build utility to build your driver, you can make sure that the resource script is converted to a resource file and attached to your driver image simply by including the name of the resource script in the SOURCES variable for the driver. For more information about the Resource Compiler, see the Windows SDK documentation. For information about using the Build utility to build your driver, see Building a Driver.

Registering as a Source of Error Messages

6/25/2019 • 2 minutes to read • Edit Online

Drivers register the source of error messages in the registry. Drivers must set two keys under **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\EventLog\System***DriverName*:

EventMessageFile (REG_EXPAND_SZ)
A list of error message sources separated by semicolons. If the driver uses standard error types, this list must include iologmsg.dll. If the driver uses error messages attached to the driver image, this must include the name of the driver image.

TypesSupported (REG_DWORD)
A bitmask of the possible severity levels that can be logged. Drivers typically set this to 7 to indicate they may log all severity levels.

For a description of how to set these registry keys from the driver's INF file, see **Registering for Event Logging**.

Writing a Bug Check Reason Callback Routine

6/28/2019 • 8 minutes to read • Edit Online

A driver can optionally provide a *KBUGCHECK_REASON_CALLBACK_ROUTINE* callback function, which the system calls after a crash dump file is written.

> **NOTE**
> This article describes the bug check *reason* callback routine, and not the *KBUGCHECK_CALLBACK_ROUTINE* callback function.

In this callback, the driver can:

- Add driver-specific data to the crash dump file
- Reset the device to a known state

Use the following routines to register and remove the callback:

- **KeRegisterBugCheckReasonCallback**
- **KeDeregisterBugCheckReasonCallback**

This callback type is overloaded, with behavior changing based on the **KBUGCHECK_CALLBACK_REASON** constant value provided at registration. This article describes the different usage scenarios.

For general information about bug check data, see Reading Bug Check Callback Data.

Bug Check Callback Routine Restrictions

A bug check callback routine executes at IRQL = HIGH_LEVEL, which imposes strong restrictions on what it can do.

A bug check callback routine cannot:

- Allocate memory
- Access pageable memory
- Use any synchronization mechanisms
- Call any routine that must execute at IRQL = DISPATCH_LEVEL or below

Bug check callback routines are guaranteed to run without interruption, so no synchronization is required. (If the bug check routine does use any synchronization mechanisms, the system will deadlock.)

A driver's bug check callback routine can safely use the **READ_PORT_*XXX***, **READ_REGISTER_*XXX***, **WRITE_PORT_*XXX***, and **WRITE_REGISTER_*XXX*** routines to communicate with the driver's device. (For information about these routines, see Hardware Abstraction Layer Routines.)

Implementing a KbCallbackAddPages Callback Routine

A kernel-mode driver can implement a *KBUGCHECK_REASON_CALLBACK_ROUTINE* callback function of type *KbCallbackAddPages* to add one or more pages of data to a crash dump file when a bug check occurs. To register this routine with the operating system, the driver calls the **KeRegisterBugCheckReasonCallback** routine. Before the driver unloads, it must call the **KeDeregisterBugCheckReasonCallback** routine to remove the registration.

Starting with Windows 8, a registered *KbCallbackAddPages* routine is called during a kernel memory dump or a

complete memory dump. In earlier versions of Windows, a registered *KbCallbackAddPages* routine is called during a kernel memory dump, but not during a complete memory dump. By default, a kernel memory dump includes only the physical pages that are being used by the Windows kernel at the time that the bug check occurs, whereas a complete memory dump includes all of the physical memory that is used by Windows. A complete memory dump does not, by default, include physical memory that is used by the platform firmware.

Your *KbCallbackAddPages* routine can supply driver-specific data to add to the dump file. For example, for a kernel memory dump, this additional data can include physical pages that are not mapped to the system address range in virtual memory but that contain information that can help you to debug your driver. The *KbCallbackAddPages* routine might add to the dump file any driver-owned physical pages that are unmapped or that are mapped to user-mode addresses in virtual memory.

When a bug check occurs, the operating system calls all the registered *KbCallbackAddPages* routines to poll drivers for data to add to the crash dump file. Each call adds one or more pages of contiguous data to the crash dump file. A *KbCallbackAddPages* routine can supply either a virtual address or a physical address for the starting page. If more than one page is supplied during a call, the pages are contiguous in either virtual or physical memory, depending on whether the starting address is virtual or physical. To supply noncontiguous pages, the *KbCallbackAddPages* routine can set a flag in the **KBUGCHECK_ADD_PAGES** structure to indicate that it has additional data and has to be called again.

Unlike a *KbCallbackSecondaryDumpData* routine, which appends data to the secondary crash dump region, a *KbCallbackAddPages* routine adds pages of data to the primary crash dump region. During debugging, primary crash dump data is easier to access than secondary crash dump data.

The operating system fills in the **BugCheckCode** member of the **KBUGCHECK_ADD_PAGES** structure that *ReasonSpecificData* points to. The *KbCallbackAddPages* routine must set the values of the **Flags**, **Address**, and **Count** members of this structure.

Before the first call to *KbCallbackAddPages*, the operating system initializes **Context** to **NULL**. If the *KbCallbackAddPages* routine is called more than once, the operating system preserves the value that the callback routine wrote to the **Context** member in the previous call.

A *KbCallbackAddPages* routine is very restricted in the actions it can take. For more information, see Bug Check Callback Routine Restrictions.

Implementing a KbCallbackDumpIo Callback Routine

A kernel-mode driver can implement a *KBUGCHECK_REASON_CALLBACK_ROUTINE* callback function of type *KbCallbackDumpIo* to perform work each time data is written to the crash dump file. The system passes, in the *ReasonSpecificData* parameter, a pointer to a **KBUGCHECK_DUMP_IO** structure. The **Buffer** member points to the current data, and the **BufferLength** member specifies its length. The **Type** member indicates the type of data currently being written, such as dump file header information, memory state, or data provided by a driver. For a description of the possible types of information, see the **KBUGCHECK_DUMP_IO_TYPE** enumeration.

The system can write the crash dump file either sequentially, or out of order. If the system is writing the crash dump file sequentially, then the **Offset** member of *ReasonSpecificData* is -1; otherwise, **Offset** is set to the current offset, in bytes, in the crash dump file.

When the system writes the file sequentially, it calls each *KbCallbackDumpIo* routine one or more times when writing the header information (**Type** = **KbDumpIoHeader**), one or more times when writing the main body of the crash dump file (**Type** = **KbDumpIoBody**), and one or more times when writing the secondary dump data (**Type** = **KbDumpIoSecondaryDumpData**). Once the system has completed writing the crash dump file, it calls the callback with **Buffer** = **NULL**, **BufferLength** = 0, and **Type** = **KbDumpIoComplete**.

The main purpose of a *KbCallbackDumpIo* routine is to allow system crash dump data to be written to devices other than the disk. For example, a device that monitors system state can use the callback to report that the system

has issued a bug check, and to provide a crash dump for analysis.

Use **KeRegisterBugCheckReasonCallback** to register a *KbCallbackDumpIo* routine. A driver can subsequently remove the callback by using the **KeDeregisterBugCheckReasonCallback** routine. If the driver can be unloaded, it must remove any registered callbacks in its *DRIVER_UNLOAD* callback function.

A *KbCallbackDumpIo* routine is strongly restricted in the actions it can take. For more information, see Bug Check Callback Routine Restrictions.

Implementing a KbCallbackSecondaryDumpData Callback Routine

A kernel-mode driver can implement a *KBUGCHECK_REASON_CALLBACK_ROUTINE* callback function of type *KbCallbackSecondaryDumpData* to provide data to append to the crash dump file.

The system sets the **InBuffer**, **InBufferLength**, **OutBuffer**, and **MaximumAllowed** members of the **KBUGCHECK_SECONDARY_DUMP_DATA** structure that *ReasonSpecificData* points to. The **MaximumAllowed** member specifies the maximum amount of dump data the routine can provide.

The value of the **OutBuffer** member determines whether the system is requesting the size of the driver's dump data, or the data itself, as follows:

- If the **OutBuffer** member of KBUGCHECK_SECONDARY_DUMP_DATA is **NULL**, the system is only requesting size information. The *KbCallbackSecondaryDumpData* routine fills in the **OutBuffer** and **OutBufferLength** members.
- If the **OutBuffer** member of KBUGCHECK_SECONDARY_DUMP_DATA equals the **InBuffer** member, the system is requesting the driver's secondary dump data. The *KbCallbackSecondaryDumpData* routine fills in the **OutBuffer** and **OutBufferLength** members, and writes the data to the buffer specified by **OutBuffer**.

The **InBuffer** member of KBUGCHECK_SECONDARY_DUMP_DATA points to a small buffer for the routine's use. The **InBufferLength** member specifies the size of the buffer. If the amount of data to be written is less than **InBufferLength**, the callback routine can use this buffer to supply the crash dump data to the system. The callback routine then sets **OutBuffer** to **InBuffer** and **OutBufferLength** to the actual amount of data written to the buffer.

A driver that must write an amount of data that is larger than **InBufferLength** can use its own buffer to provide the data. This buffer must have been allocated before the callback routine is executed, and must reside in resident memory (such as nonpaged pool). The callback routine then sets **OutBuffer** to point to the driver's buffer, and **OutBufferLength** to the amount of data in the buffer to be written to the crash dump file.

Each block of data to be written to the crash dump file is tagged with the value of the **Guid** member of the **KBUGCHECK_SECONDARY_DUMP_DATA** structure. The GUID used must be unique to the driver. To display the secondary dump data corresponding to this GUID, you can use the **.enumtag** command or the **IDebugDataSpaces3::ReadTagged** method in a debugger extension. For information about debuggers and debugger extensions, see Windows Debugging.

A driver can write multiple blocks with the same GUID to the crash dump file, but this is very poor practice, because only the first block will be accessible to the debugger. Drivers that register multiple *KbCallbackSecondaryDumpData* routines should allocate a unique GUID for each callback.

Use **KeRegisterBugCheckReasonCallback** to register a *KbCallbackSecondaryDumpData* routine. A driver can subsequently remove the callback routine by using the **KeDeregisterBugCheckReasonCallback** routine. If the driver can be unloaded, then it must remove any registered callback routines in its *DRIVER_UNLOAD* callback function.

A *KbCallbackSecondaryDumpData* routine is very restricted in the actions it can take. For more information, see Bug Check Callback Routine Restrictions.

Implementing a KbCallbackTriageDumpData Callback Routine

Starting in Windows 10, version 1809 and Windows Server 2019, a kernel-mode driver can implement a *KBUGCHECK_REASON_CALLBACK_ROUTINE* callback function of type *KbCallbackTriageDumpData* to add virtual memory ranges to a carved minidump file. The system passes, in the *ReasonSpecificData* parameter, a pointer to a **KBUGCHECK_TRIAGE_DUMP_DATA** structure that describes the dump data.

In the following example, the driver configures a triage dump array and then registers a minimal implementation of the callback:

```
// Globals

KBUGCHECK_REASON_CALLBACK_RECORD ExampleBugcheckCallbackRecord;
PKTRIAGE_DUMP_DATA_ARRAY gTriageDumpDataArray;

//  call this register function from DriverInit, etc.

VOID ExampleRegisterTriageDataCallbacks()
{

    //
    // Allocate a triage dump array in the non-paged pool.
    //

gTriageDumpDataArray =
    (PKTRIAGE_DUMP_DATA_ARRAY)ExAllocatePoolWithTag(NonPagedPoolNx, 2*PAGE_SIZE, "Xmpl");

    //
    // Initialize the dump data block array.
    //

    KeInitializeTriageDumpDataArray( gTriageDumpDataArray, 2*PAGE_SIZE, "Example");

    KeInitializeCallbackRecord( &ExampleBugcheckCallbackRecord );

    KeRegisterBugCheckReasonCallback(
        &ExampleBugCheckCallbackRecord,
        ExampleBugCheckCallbackRoutine,
        KbCallbackTriageDumpData,
        "Example"
        );
}

// Callback function

VOID
ExampleBugCheckCallbackRoutine(
    KBUGCHECK_CALLBACK_REASON Reason,
    PKBUGCHECK_REASON_CALLBACK_RECORD Record,
    PVOID Data,
    ULONG Length
    )
{
    PKBUGCHECK_TRIAGE_DUMP_DATA DumpData;
    NTSTATUS Status;

    DumpData = (PKBUGCHECK_TRIAGE_DUMP_DATA) Data;

    Status = KeAddTriageDumpDataBlock(gTriageDumpDataArray, gImportant, SizeofGImportant);

    // Pass our arrays back

    if (NT_SUCCESS(Status)) {
        DumpData->RequiredDataArray = gTriageDumpDataArray;
    }

    return;
}
```

A *KbCallbackTriageDumpData* routine is very restricted in the actions it can take. For more information, see Bug Check Callback Routine Restrictions.

Using Safe String Functions

6/25/2019 • 2 minutes to read • Edit Online

Many system security problems are caused by poor buffer handling and the resulting buffer overruns. Poor buffer handling is often associated with string manipulation operations. The standard string manipulation functions that are supplied by C/C++ language runtime libraries (**strcat**, **strcpy**, **sprintf**, and so on) do not prevent writing beyond the end of buffers.

Two new sets of string manipulation functions, called *safe string functions*, provide additional processing for proper buffer handling in your code. These safe string functions are available in the Windows Driver Kit (WDK) and for Microsoft Windows XP SP1 and later versions of the Driver Development Kit (DDK) and Windows SDK. They are intended to replace their built-in C/C++ counterparts and similar routines that are supplied by Windows.

One set of safe string functions are for use in kernel-mode code. These functions are prototyped in a header file named Ntstrsafe.h. This header file and an associated library are available in the WDK.

The other set of safe string functions are for use in user-mode applications. A corresponding header file, Strsafe.h, contains prototypes for these functions. That file and an associated library are available in the Windows SDK. For more information about Strsafe.h, see Using the Strsafe.h Functions.

The set of kernel-mode safe string functions consists of the following two subsets:

- Safe string functions for Unicode and ANSI characters

  Each of these functions is available in a W-suffixed version that supports double-byte Unicode characters and an A-suffixed version that supports single-byte ANSI characters. For example, **RtlStringCbCatN**, which concatenates two strings and limits the length of the appended string, is available as **RtlStringCbCatNW** and **RtlStringCbCatNA**.

- Safe string functions for UNICODE_STRING structures

  Each of these functions accepts a **UNICODE_STRING** structure as an input or output parameter or both. For example, **RtlStringCbCopyUnicodeString** accepts the structure as an input parameter, **RtlUnicodeStringCopyString** accepts the structure as an output parameter, and **RtlUnicodeStringCopy** accepts the structure as both an input and output parameter.

The kernel-mode safe string functions provide the following features:

- Each safe string function receives the size of the destination buffer as input. The function can thus ensure that it does not write past the end of the buffer.
- The Unicode and ANSI string functions terminate all output strings with a NULL character, even if the operation truncates the intended result.
- All safe string functions return an NTSTATUS value, with only one possible success code (STATUS_SUCCESS).
- Most safe string functions are available in both a byte-counted and a character-counted version. For example, **RtlStringCbCat** concatenates two byte-counted strings and **RtlStringCchCat** concatenates two character-counted strings.
- Most safe string functions are available in an extended, Ex-suffixed version that provides additional functionality. For example, **RtlStringCbCatEx** extends the functionality of **RtlStringCbCat**.

This section includes the following topics:

Summary of Kernel-Mode Safe String Functions

6/25/2019 • 2 minutes to read • Edit Online

The following table summarizes the safe string functions that are available to kernel-mode drivers, and it indicates the C/C++ language runtime library functions that they replace. If a function's name contains **Cb**, the function treats strings as byte-counted. If a function's name contains **Cch**, the function treats strings as character-counted.

FUNCTIONS	PURPOSE	REPLACES
RtlStringCbCatA **RtlStringCbCatExA** **RtlStringCchCatA** **RtlStringCchCatExA** **RtlUnicodeStringCatString** **RtlUnicodeStringCatStringEx** **RtlUnicodeStringCbCatStringN** **RtlUnicodeStringCchCatStringN** **RtlUnicodeStringCbCatStringNEx** **RtlUnicodeStringCchCatStringNEx**	Concatenate two strings.	**strcat** **wcscat**

FUNCTIONS	PURPOSE	REPLACES
trin gCb Cat Stri ngN RtlU nico deS trin gCb Cat Stri ngN Ex RtlU nico deS trin gCc hCa tStri ngN RtlU nico deS trin gCc hCa tStri ngN Ex		

FUNCTIONS	PURPOSE	REPLACES
RtlStringCbCatN **RtlStringCbCatNEx** **RtlStrinCchCatN** **RtlStrinCchCatNEx** **RtlUnicodeStrinCbCatN** **RtlUnicodeStrinCbCatNEx** **RtlUnicodeStrinCchCatN** **RtlUnicodeStrinCchCatNEx**	Concatenate two byte-counted strings, while limiting the size of the appended string.	**strncat** **wcsncat**
RtlStringCbCopy **RtlStringCbCopyEx** **RtlStringCbCopyN** **RtlStringCbCopyNEx** **RtlStringCchCopy** **RtlStringCchCopyEx** **RtlStringCchCopyN** **RtlStringCchCopyNEx** **RtlUnicodeStringCbCopyN** **RtlUnicodeStringCbCopyNEx** **RtlUnicodeStringCchCopyN** **RtlUnicodeStringCchCopyNEx**	Copy a string into a buffer.	**strcpy** **wcscpy**

FUNCTIONS	PURPOSE	REPLACES
CopyExRtlStringCbCopyUnicodeStringRtlStringCbCopyUnicodeStringExRtlStringCchCopyRtlStringCchCopyExRtlStringCchCopyUnicodeStringRtlStringCchCopyUnicodeStringExRtlUnicodeStringCopyRtlUnicodeStringCopyExRtlUnicodeStringCopyUnicodeStringRtlUnicodeStringCopyUnicodeStringEx		

FUNCTIONS	PURPOSE	REPLACES
trin gCo pyS trin g RtlU nico deS trin gCo pyS trin gEx		
RtlS trin gCb Cop yN RtlS trin gCb Cop yNE x RtlS trin gCc hCo pyN RtlS trin gCc hCo pyN Ex RtlU nico deS trin gCb Cop yN RtlU nico deS trin gCb Cop yNE x RtlU nico deS trin gCc hCo pyN RtlU nico deS trin gCc hCo	Copy a string into a buffer, while limiting the size of the copied string.	strn cpy wcs ncp y

FUNCTIONS	PURPOSE	REPLACES
pyNExRtlUnicodeStringCbCopyStringN RtlUnicodeStringCbCopyStringNExRtlUnicodeStringCchCopyStringN RtlUnicodeStringCchCopyStringNEx		

FUNCTIONS	PURPOSE	REPLACES
RtlStringCbLength **RtlStringCchLength** **RtlUnicodeStringCbLength** **RtlUnicodeStringCchLength**	Determine the length of a supplied string.	**strlen** **wcslen**

FUNCTIONS	PURPOSE	REPLACES
RtlS trin gCb Prin tf **RtlS trin gCb Prin tfEx** **RtlS trin gCc hPri ntf** **RtlS trin gCc hPri ntfE x** **RtlU nico deS trin gPri ntf** **RtlU nico deS trin gPri ntfE x**	Create a formatted text string that is based on a format string and a set of additional function arguments.	**spri ntf** **swp rintf** **_sn prin tf** **_sn wpr intf**

FUNCTIONS	PURPOSE	REPLACES
RtlS **trin** **gCb** **VPri** **ntf** **RtlS** **trin** **gCb** **VPri** **ntfE** **x** **RtlS** **trin** **gCc** **hVP** **rintf** **RtlS** **trin** **gCc** **hVP** **rintf** **Ex** **RtlU** **nico** **deS** **trin** **gVP** **rintf** **RtlU** **nico** **deS** **trin** **gVP** **rintf** **Ex**	Create a formatted text string that is based on a format string and one additional function argument.	**vsp** **rintf** **vsw** **prin** **tf** **_vsn** **prin** **tf** **_vsn** **wpr** **intf**

FUNCTIONS	PURPOSE	REPLACES
RtlUnicodeStringInit **RtlUnicodeStringInitEx** **RtlUnicodeStringValidate** **RtlUnicodeStringValidateEx**	Initialize or validate a **UNICODE_STRING** structure.	None

Importing Kernel-Mode Safe String Functions

6/25/2019 • 2 minutes to read • Edit Online

Starting with Windows XP, the kernel-mode safe string library is available as a collection of inline functions that are defined in the Ntstrsafe.h header file.

To use the kernel-mode safe string functions

Include the header file, as shown.

```
#include <ntstrsafe.h>
```

You can make available only the byte-counted or only the character-counted safe string functions.

To allow only byte-counted functions

Include the following line in your code before including the Ntstrsafe.h header file.

```
#define NTSTRSAFE_NO_CCH_FUNCTIONS
```

To allow only character-counted functions

Include the following line in your code before including the Ntstrsafe.h header file.

```
#define NTSTRSAFE_NO_CB_FUNCTIONS
```

You can define either NTSTRSAFE_NO_CB_FUNCTIONS or NTSTRSAFE_NO_CCH_FUNCTIONS, but not both.

You can make the **UNICODE_STRING** structure functions unavailable.

To make UNICODE_STRING structure functions unavailable

Include the following line in your code before including the Ntstrsafe.h header file.

```
#define NTSTRSAFE_NO_UNICODE_STRING_FUNCTIONS
```

The maximum number of characters that any ANSI or Unicode string can contain is NTSTRSAFE_MAX_CCH. The maximum number of characters that a **UNICODE_STRING** structure can contain is NTSTRSAFE_UNICODE_STRING_MAX_CCH. These constants are defined in Ntstrsafe.h.

Your driver can assign smaller values to NTSTRSAFE_MAX_CCH and NTSTRSAFE_UNICODE_STRING_MAX_CCH by including the following lines in your code before including Ntstrsafe.h.

```
#define NTSTRSAFE_MAX_CCH  <new-value>
#define NTSTRSAFE_UNICODE_STRING_MAX_CCH  <new-value>
```

Directives in Ntstrsafe.h verify that your new values are not larger than the default values.

Using Safe Integer Functions

12/5/2018 • 2 minutes to read • Edit Online

One way to minimize security problems is to prevent integer overflows and underflows. Integer overflows occur when the result of an arithmetic operation is larger than the memory space of the data type that is set to receive it. This results in the truncation of the integer and an incorrect result. An underflow occurs when an operation, usually subtraction, gives an incorrect result. Casting between two data types can also cause incorrect results due to truncation of a result that does not fit the new memory space.

The ntintsafe library provides a set of C functions that perform safe integer arithmetic operations with bounds checking to prevent overflows and underflows in kernel-mode code. These functions correspond to the Windows IntSafe functions that are used by application code. You use these functions to calculate an index or buffer size, or to compute some other form of bounds check. The functions are optimized for speed.

Safe integer functions offer the following advantages:

- The size of the destination buffer is always provided to the function to ensure that the function does not write past the end of the buffer.
- Buffers are guaranteed to be null-terminated, even if the operation truncates the intended result.
- All functions return an NTSTATUS, with only one possible success code (STATUS_SUCCESS) and one possible error condition (STATUS_INTEGER_OVERFLOW).

The ntintsafe library has two categories of functions:

- **Conversion functions**—These functions perform conversions between two data types.
- **Arithmetic functions**—These functions perform addition, subtraction, and multiplication operations for each data type. There are no division operations because there are no overflow conditions.

Summary of Kernel-Mode Safe Integer Functions

Importing Kernel-Mode Safe Integer Functions

Summary of Kernel-Mode Safe Integer Functions

6/25/2019 • 2 minutes to read • Edit Online

The following table summarizes the safe integer functions that are available to kernel-mode drivers.

FUNCTIONS	PURPOSE
RtlDWordPtrAdd **RtlIntPtrAdd** **RtlLongPtrAdd** **RtlPtrdiffTAdd** **RtlShortAdd** **RtlSizeTAdd** **RtlSSIZETAdd** **RtlUInt8Add** **RtlUIntAdd** **RtlUIntPtrAdd** **RtlULongAdd**	Addition Functions

FUNCTIONS	PURPOSE
RtlULongLongAdd RtlULongPtrAdd RtlUShortAdd	
RtlDWordPtrMult RtlInt8Mult RtlIntMult RtlIntPtrMult RtlLongLongMult RtlLongMult RtlLongPtrMult RtlPtrdiffTMult RtlShortMult RtlSSIZETMult RtlUInt8Mult	Multiplication Functions

FUNCTIONS	PURPOSE
RtlSizeTMult RtlULongMult RtlULongLongMult RtlUIntPtrMult RtlUIntMult RtlULongPtrMult RtlUShortMult	
RtlShortSub RtlUShortSub RtlULongPtrSub RtlULongLongSub RtlULongSub RtlUInt8Sub RtlUIntPtrSub RtlUIntSub RtlS	Subtraction Functions

FUNCTIONS	PURPOSE
SIZE TSub RtlSizeTSub RtlDWordPtrSub RtlInt8Sub RtlPtrdiffTSub RtlLongSub RtlIntSub RtlLongLongSub RtlIntPtrSub RtlLongPtrSub	

FUNCTIONS	PURPOSE
RtlUShortToShort RtlLongPtrToShort RtlLongToShort RtlLongLongToShort RtlULongLongToShort RtlULongToShort RtlIntPtrToShort RtlULongPtrToShort RtlIntToShort	Conversion to Short
RtlLongPtrToChar	Conversion to Char

FUNCTIONS	PURPOSE
ar RtlLong LongTo Char RtlUInt8 ToChar RtlULongTo Char RtlLong ToChar RtlULong gLong ngTo Char RtlUInt oChar RtlI ntTo Char RtlI ntPt rTo Char RtlULong gPt rTo Char RtlS hort ToC har RtlU Sho rtTo Char RtlB yte ToC har RtlU IntP trTo Char	

FUNCTIONS	PURPOSE
RtlIntPtrToInt **RtlLongLongToInt** **RtlLongPtrToInt** **RtlULongLongToInt** **RtlULongPtrToInt** **RtlULongToInt** **RtlUIntToInt** **RtlUIntPtrToInt**	Conversion to Int

FUNCTIONS	PURPOSE
RtlUInt8ToInt8 RtlLongLongToInt8 RtlLongPtrToInt8 RtlULongLongPtrToInt8 RtlULongLongToInt8 RtlULongLongToLongLongToInt8 RtlULongToInt8 RtlIntPtrToInt8 RtlIntToInt8 RtlUIntPtrToInt8 RtlUIntToInt8 RtlUShortToInt8 RtlShortToInt8	Conversion to Int8

FUNCTIONS	PURPOSE
RtlUIntPtrToInt16	Conversion to Int16
RtlLongLongToIntPtr RtlLongPtrToIntPtr RtlLongToIntPtr RtlULongToIntPtr RtlULongPtrToIntPtr RtlUIntPtrToIntPtr RtlUIntToIntPtr	Conversion to IntPtr

FUNCTIONS	PURPOSE
RtlULongToLong **RtlIntPtrToLong** **RtlULongLongToULongPtr** **RtlULongPtrToLong**	Conversion to Long

FUNCTIONS	PURPOSE
RtlIntPtrToLongPtr **RtlULongToLongPtr** **RtlULongPtrToLongPtr** **RtlUIntPtrToLongPtr** **RtlULongLongToLongPtr** **RtlULongPtrToLongLongPtr** **RtlUIntToLongPtr** **RtlULongPtrToLongPtr**	Conversion to LongPtr

FUNCTIONS	PURPOSE
RtlU **Lon** **gLo** **ngT** **oLo** **ngL** **ong** **RtlU** **Lon** **gPt** **rTo** **Lon** **gLo** **ng** **RtlU** **IntP** **trTo** **Lon** **gLo** **ng**	Conversion to LongLong

FUNCTIONS	PURPOSE
RtlIntPtrToUShort RtlInt8ToUShort RtlULongToUShort RtlUIntToUShort RtlLongLongToUShort RtlLongPtrToUShort RtlLongToUShort RtlSHortToUShort RtlUIntPtrToUShort RtlUIntToUShort RtlULongLongToUShort RtlULongPtrToUShort	Conversion to UShort

FUNCTIONS	PURPOSE
RtlUShortToUChar RtlInt8ToUChar RtlIntPtrToUChar RtlIntToUChar RtlLongLongToUChar RtlLongPtrToUChar RtlLongToUChar RtlShortToUChar RtlUIntPtrToUChar RtlUIntToUChar RtlULongLongToUChar RtlULongPtrToUChar RtlULon	Conversion to UChar

FUNCTIONS	PURPOSE
RtlUCharToUChar	
RtlInt8ToUInt RtlULongToUInt RtlULongToUIntPtrToUInt RtlUShortToUInt RtlLongPtrToUIntPtr RtlLongToUInt RtlULongToUInt RtlULongLongToUIntPtrToUInt RtlUIntPtrToUIntPtr RtlULongLongToUInt RtlULongPtrToUIntPtr	Conversion to Uint
RtlUShortToUInt8 RtlI	Conversion to Uint8

FUNCTIONS	PURPOSE
nt8 ToU Int8 RtlL ong Lon gTo UInt 8 Rtl ntT oUl nt8 Rtl ntPt rTo UInt 8 RtlL ong PtrT oUl nt8 RtlS hort ToU Int8 RtlL ong ToU Int8 RtlU IntP trTo UInt 8 RtlU IntT oUl nt8 RtlU Lon gLo ngT oUl nt8 RtlU Lon gPt rTo UInt 8 RtlU Lon gTo UInt 8	
RtlU IntP trTo UInt 16	Conversion to Uint16

FUNCTIONS	PURPOSE
RtlULongToUIntPtr RtlLongToUIntPtr RtlShortToUIntPtr RtlInt8ToUIntPtr RtlIntPtrToUIntPtr RtlLongPtrToUIntPtr RtlULongLongToUIntPtr RtlULongPtrToUIntPtr	Conversion to UintPtr

FUNCTIONS	PURPOSE
RtlULongPtrToULong RtlInt8ToULong RtlIntPtrToULong RtlIntToULong RtlLongToULong RtlPtrToULong RtlLongToULong RtlShortToULong RtlUIntPtrToULong RtlULongLongToULong	Conversion to ULong

FUNCTIONS	PURPOSE
RtlShortToULongLong RtlInt8ToULongLong RtlIntToULongLong RtlIntPtrToULongLong RtlLongToULongLong RtlPtrToULongLong RtlULongToULongLong	Conversion to ULongLong

FUNCTIONS	PURPOSE
RtlULongLongToULongPtr **RtlIntPtrToULongPtr** **RtlLongPtrToULongPtr** **RtlInt8ToULongPtr** **RtlLongToULongPtr** **RtlShortToULongPtr**	Conversion to ULongPtr

Importing Kernel-Mode Safe Integer Functions

12/5/2018 • 2 minutes to read • Edit Online

The kernel-mode safe integer functions are available as inline code that is contained in ntintsafe.h or in a library that you link your code to. This header file is available in the Windows Driver Kit (WDK).

It is important to note that you must use arithmetic operations on unsigned values. To use a signed value, you must use a conversion function to first convert the signed value to an unsigned value safely before using the arithmetic function.

To use the inline versions of the kernel-mode safe integer functions

Include the header file, as shown.

```
#include <ntintsafe.h>
```

Determining Whether the Operating System Is Running in Safe Mode

12/5/2018 • 2 minutes to read • Edit Online

This topic describes how a device driver can determine whether the operating system that it is running on was started in Safe Mode. This topic also describes how to prevent a driver from operating in Safe Mode.

The Microsoft Windows operating system kernel exports a pointer named **InitSafeBootMode**. **InitSafeBootMode** points to a ULONG variable that contains the Safe Mode settings that are currently in effect. A device driver can examine these settings to determine whether the operating system is running in Safe Mode.

The following table lists the modes for values of the **InitSafeBootMode** variable.

VALUE	MODE
0	The operating system is not in Safe Mode.
1	SAFEBOOT_MINIMAL
2	SAFEBOOT_NETWORK
3*	SAFEBOOT_DSREPAIR

Note * The value 3 applies to Windows domain controllers only.

To use the **InitSafeBootMode** variable, you must declare it in your driver, as the following code example shows.

```
extern PULONG InitSafeBootMode;
```

After you declare **InitSafeBootMode**, you can use the following code example to determine whether the operating system is running in Safe Mode.

```
if (*InitSafeBootMode > 0) {
    // The operating system is in Safe Mode.
    // Take appropriate action.
    //
}
```

To prevent a driver from operating in Safe Mode, use the technique in the following list that matches your driver type:

- **Function drivers**

  If your function driver has a service start type of SERVICE_BOOT_START, check the value of **InitSafeBootMode** in the function driver's *AddDevice* routine. If the system is in Safe Mode, return a failure status.

  Note You must never return failure from the **DriverEntry** routine.

- **Filter drivers**

  If your filter driver starts during system startup, check the value of **InitSafeBootMode** in the filter driver's *AddDevice* routine. If the operating system is in Safe Mode, do the following:

  1. Do not attach the filter device object to the device stack.
  2. Return success from the filter driver's *AddDevice* routine.

- **Other drivers**

  For drivers other than function or filter drivers, check the value of **InitSafeBootMode** in the driver's **DriverEntry** routine. If the operating system is in Safe Mode, return a failure status.

Using GUIDs in Drivers

6/25/2019 • 2 minutes to read • Edit Online

Drivers and other system components use *globally unique identifiers* (GUIDs) to identify a variety of items. System components define GUIDs for items such as device setup classes, PnP events, WMI events, and still image events. Driver writers can create GUIDs for items such as device interface classes, custom PnP events, and custom WMI events. Drivers and applications include header files that define the GUIDs that they use.

This section includes the following topics:

For information about using GUIDs in user-mode applications, see Microsoft Windows SDK documentation.

Defining and Exporting New GUIDs

6/25/2019 • 2 minutes to read • Edit Online

You define a new GUID for an item the driver exports to other system components, drivers, or applications. For example, you define a new GUID for a custom PnP event on one of its devices. To define and export a new GUID, you must do the following:

1. Choose a symbolic name for the GUID.

   Choose a name that represents the purpose of the GUID. For example, the operating system uses such names as GUID_BUS_TYPE_PCI and PARPORT_WMI_ALLOCATE_FREE_COUNTS_GUID.

2. Generate a value for the GUID using Uuidgen.exe or Guidgen.exe. When you install the Microsoft Windows SDK, Uuidgen.exe is automatically installed. Guidgen.exe is available from the Microsoft Exchange Server GUID Generator download page.

   These utilities generate a unique, formatted string that represents a 128-bit value. The "-s" switch on Uuidgen.exe outputs the GUID formatted as a C structure.

3. Define the GUID in an appropriate header file.

   Use the **DEFINE_GUID** macro (defined in Guiddef.h) to associate the GUID symbolic name with its value (see Example 1).

   Example 1: Defining GUIDs in a GUID-Only Header File

```
:

DEFINE_GUID( GUID_BUS_TYPE_PCMCIA, 0x09343630L, 0xaf9f, 0x11d0,
    0x92,0x9f, 0x00, 0xc0, 0x4f, 0xc3, 0x40, 0xb1 );
DEFINE_GUID( GUID_BUS_TYPE_PCI, 0xc8ebdfb0L, 0xb510, 0x11d0,
    0x80,0xE9, 0x00, 0x00, 0xf8, 0x1e, 0x1b, 0x30 );

:
```

   If the GUID is defined in a header file that contains statements other than GUID definitions, you must take an extra step to ensure that the GUID is instantiated in drivers that include the header file. The **DEFINE_GUID** statement must occur outside any **#ifdef** statements that prevent multiple inclusion. Otherwise, if the header file is included in a precompiled header, the GUID will not be instantiated in drivers that use the header file. See Example 2 for a sample GUID definition in a mixed header file.

   Example 2: Defining GUIDs in a Mixed Header File

```
#ifndef _NTDDSER_    // this ex. is from a serial driver .h file
#define _NTDDSER_

:
// Put other header file definitions here.
:

#endif  // _NTDDSER_

#ifdef DEFINE_GUID   // Do not break compiles of drivers that
                     // include this header but that do not
                     // want the GUIDs.
//
// Put GUID definitions outside of the multiple inclusion
// protection.

DEFINE_GUID(GUID_CLASS_COMPORT, 0x86e0d1e0L, 0x8089, 0x11d0, 0x9c,
    0xe4, 0x08, 0x00, 0x3e, 0x30, 0x1f, 0x73);

DEFINE_GUID (GUID_SERENUM_BUS_ENUMERATOR, 0x4D36E978, 0xE325,
    0x11CE, 0xBF, 0xC1, 0x08, 0x00, 0x2B, 0xE1, 0x03, 0x18);

:
#endif  // DEFINE_GUID
```

Putting a GUID definition outside statements that prevent multiple inclusion does not cause multiple instances of the GUID in a driver because **DEFINE_GUID** defines the GUID as an EXTERN_C variable. Multiple declarations of an EXTERN variable are allowed as long as the types match.

4. When creating a GUID for a new device setup class or device interface class, the following rules apply:

   - Do not use a single GUID to identify both a device setup class and a device interface class.
   - When creating a symbolic name to associate with the GUID, use the following convention:

     For device setup classes, use the format GUID_DEVCLASS_*XXX*.

     For device interface classes, use the format GUID_DEVINTERFACE_*XXX*.

Including GUIDs in Driver Code

12/5/2018 • 2 minutes to read • Edit Online

To use GUIDs in a kernel-mode driver, you must do two things:

1. Include the Initguid.h header file that redefines the **DEFINE_GUID** macro.

   The Initguid.h header file redefines the **DEFINE_GUID** macro to instantiate GUIDs (versus just declaring an EXTERN reference). Include this header file in the driver source file where the GUIDs should be instantiated. (User-mode applications include Objbase.h before including header files containing GUID definitions.)

2. Include the header file(s) that define the GUIDs.

   After the statement to include Initguid.h, you include the header files containing the GUID definitions. A driver might include more than one header file that contains GUID definitions, including system-supplied header files and third-party header files.

The following code excerpt shows the sequence of statements for including GUIDs:

```
:
// include system headers here such as wdm.h

#include <initguid.h>

// include system and driver-specific header files here that contain
// GUID definitions

...
```

Put the above statements in one module of the driver; typically the main module. When the above statements are present, the driver refers to a GUID using its symbolic name.

Using Floating Point in a WDM Driver

6/25/2019 • 2 minutes to read • Edit Online

Last updated

- July 2016

Kernel-mode WDM drivers for Windows must follow certain guidelines when using floating-point operations. These differ between x86 and x64 systems. By default, Windows turns off arithmetic exceptions for both systems.

x86 systems

Kernel-mode WDM drivers for x86 systems must wrap the use of floating point calculations between calls to **KeSaveExtendedProcessorState** and **KeRestoreExtendedProcessorState**. The floating point operations must be placed in a non-inline subroutine to make sure that floating point calculations are not performed before checking the return value of **KeSaveExtendedProcessorState** due to compiler reordering.

The compiler makes use of MMX/x87 also known as the floating-point unit (FPU) registers for such calculations, which can be concurrently used by a user-mode application. Failure to save these registers before using them, or failure to restore them when finished, may cause calculation errors in applications.

Drivers for x86 systems can call **KeSaveExtendedProcessorState** and perform floating point calculations at IRQL <= DISPATCH_LEVEL. Floating-point operations are not supported in interrupt service routines (ISRs) on x86 systems.

x64 systems

The 64-bit compiler does not use the MMX/x87 registers for floating point operations. Instead, it uses the SSE registers. x64 kernel mode code is not allowed to access the MMX/x87 registers. The compiler also takes care of properly saving and restoring the SSE state, therefore, calls to **KeSaveExtendedProcessorState** and **KeRestoreExtendedProcessorState** are unnecessary and floating point operations can be used in ISRs. Use of other extended processor features such as AVX, requires saving and restoring extended state. For more information see Using extended processor features in Windows drivers.

Example

The following example shows how a WDM driver should wrap its FPU access:

```
__declspec(noinline)
VOID
DoFloatingPointCalculation(
    VOID
    )
{
    double Duration;
    LARGE_INTEGER Frequency;

    Duration = 1000000.0;
    DbgPrint("%I64x\n", *(LONGLONG*)&Duration);
    KeQueryPerformanceCounter(&Frequency);
    Duration /= (double)Frequency.QuadPart;
    DbgPrint("%I64x\n", *(LONGLONG*)&Duration);
}

NTSTATUS
DriverEntry(
    _In_ PDRIVER_OBJECT DriverObject,
    _In_ PUNICODE_STRING RegistryPath
    )
{

    XSTATE_SAVE SaveState;
    NTSTATUS Status;

    Status = KeSaveExtendedProcessorState(XSTATE_MASK_LEGACY, &SaveState);
    if (!NT_SUCCESS(Status)) {
        goto exit;
    }

    __try {
        DoFloatingPointCalculation();
    }
    __finally {
        KeRestoreExtendedProcessorState(&SaveState);
    }

exit:
    return Status;
}
```

In the example, the assignment to the floating-point variable occurs between calls to **KeSaveExtendedProcessorState** and **KeRestoreExtendedProcessorState**. Because any assignment to a floating-point variable uses the FPU, drivers must ensure that **KeSaveExtendedProcessorState** has returned without error before initializing such a variable.

The preceding calls are unnecessary on an x64 system and harmless when the XSTATE_MASK_LEGACY flag is specified. Therefore, there is no need to change the code when compiling the driver for an x64 system.

On x86-based systems, the FPU is reset to its default state by a call to FNINIT, upon return from **KeSaveExtendedProcessorState**.

Using Files In A Driver

6/25/2019 • 2 minutes to read • Edit Online

The Microsoft Windows executive represents files by *file objects*, which are executive objects that are managed by the object manager. (Directories are also represented by file objects.)

Kernel-mode components refer to a file by its object name, which is **\DosDevices** concatenated to the file's full path. (On Microsoft Windows 2000 and later versions of the operating system, **\??** is equivalent to **\DosDevices**.) For example, the object name of the C:\WINDOWS\example.txt file is **\DosDevices\C:\WINDOWS\example.txt**. You use the object name to open a handle to a file. For more information about object names, see Object Names.

To use a file

1. Open a handle to the file.

   For more information, see Opening a Handle to a File.

2. Perform the intended operations by calling the appropriate **Zw*Xxx*File** routines.

   For more information, see Using a File Handle.

3. Close the handle by calling **ZwClose**.

Every time that you open a handle to a file, the Windows executive creates a file object that represents the file, and it returns an open handle to that object. Therefore, multiple file objects can exist for a single file. (Because a user-mode application can copy a handle, multiple handles can also exist for the same file object.) After all the open handles to a file object are closed, the Windows executive deletes the file object.

Opening a Handle to a File

6/25/2019 • 2 minutes to read • Edit Online

To open a handle to a file, perform the following steps:

1. Create an **OBJECT_ATTRIBUTES** structure, and call the **InitializeObjectAttributes** macro to initialize the structure. You specify the file's object name as the *ObjectName* parameter to **InitializeObjectAttributes**.
2. Open a handle to the file by passing the **OBJECT_ATTRIBUTES** structure to **IoCreateFile**, **ZwCreateFile**, or **ZwOpenFile**.

   If the file does not exist, **IoCreateFile** and **ZwCreateFile** will create it, whereas **ZwOpenFile** will return STATUS_OBJECT_NAME_NOT_FOUND.

Note that drivers almost always use **ZwCreateFile** or **ZwOpenFile** rather than **IoCreateFile**.

When you call **IoCreateFile**, **ZwCreateFile**, or **ZwOpenFile**, the Windows executive creates a new file object to represent the file, and it provides an open handle to the object. This file object persists until you close all the open handles to it.

Whichever routine you call, you must pass the access rights you need as the *DesiredAccess* parameter. These rights must cover all the operations that your driver will perform. The following table lists these operations and the corresponding access right to request.

OPERATION	REQUIRED ACCESS RIGHT
Read from the file.	FILE_READ_DATA or GENERIC_READ
Write to the file.	FILE_WRITE_DATA or GENERIC_WRITE
Write only to the end of the file.	FILE_APPEND_DATA
Read the file's metadata, such as the file's creation time.	FILE_READ_ATTRIBUTES or GENERIC_READ
Write the file's metadata, such as the file's creation time.	FILE_WRITE_ATTRIBUTES or GENERIC_WRITE

For more information about the values available for *DesiredAccess*, see **ZwCreateFile**.

Using a File Handle

6/25/2019 • 2 minutes to read • Edit Online

The following table lists the operations that drivers can perform on a file handle and the corresponding routines that carry out those operations.

OPERATION	ROUTINE TO CALL
Read data from the file.	**ZwReadFile**
Write data to the file.	**ZwWriteFile**
Read metadata for the file or file handle.	**ZwQueryInformationFile**
Write metadata for the file or file handle.	**ZwSetInformationFile**

To indicate where in the file to begin reading or writing data, you pass a *ByteOffset* parameter to **ZwReadFile** or **ZwWriteFile**, respectively.

If you opened the handle with FILE_APPEND_DATA access, all data is written to the end of the file, and the *ByteOffset* parameter is ignored.

Under certain conditions, the I/O manager maintains a current file-position pointer for the file. You can begin a read or write operation at that position by specifying **NULL** for *ByteOffset*. For more information about when the current file-position pointer exists, see Using the Current File Position later in this section.

To examine or change information about a file, call **ZwQueryInformationFile** or **ZwSetInformationFile**, respectively. You specify the particular type of information as the *FileInformationClass* parameter to each routine. For example, setting *FileInformationClass* to **FileBasicInformation** allows you to examine or change a **FILE_BASIC_INFORMATION** structure, which contains members for the file-creation time and the last-access time, among others. For information about all the possible values for *FileInformationClass*, see **FILE_INFORMATION_CLASS**.

Using the Current File Position

6/25/2019 • 2 minutes to read • Edit Online

When you create or open a file, you can cause the I/O manager to create a current file-position pointer that is associated with the file handle. Once you have done so, you can read and write data to the current file position, and the I/O manager will automatically update the position by the number of bytes that were read or written.

By default, the I/O manager does not maintain a current file-position pointer. This default provides efficiency—because correctly maintaining the current file position requires the I/O manager to synchronize every read and write operation on the file object.

To create a handle that has an associated current file-position pointer, specify the SYNCHRONIZE access right in the *DesiredAccess* parameter to **ZwCreateFile**, **IoCreateFile**, or **ZwOpenFile**, and either FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT in the *CreateOptions* or *OpenOptions* parameter. Be sure that you do not also specify the FILE_APPEND_DATA access right.

ZwReadFile and **ZwWriteFile** automatically update the current file-position pointer so that it points just beyond the data affected by the operation. For example, if you read 20 bytes starting at byte offset 101, **ZwReadFile** will update the current file position to 121.

You can examine or change the current file position by calling **ZwQueryInformationFile** or **ZwSetInformationFile**, respectively. In either case, set the *FileInformationClass* parameter to **FilePositionInformation**.

Registry Key Object Routines

6/25/2019 • 2 minutes to read • Edit Online

The Windows executive represents registry keys as executive objects that are managed by the object manager. (For more information about the object manager, see Object Management.) In particular, every key has an object name, and you can open a handle to a key.

User-mode applications access keys relative to global handles, such as HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER. However, these handles are not available to kernel-mode code. Instead, you refer to a key by its object name. The root for all registry keys is the **\Registry** object. The global handles correspond to descendants of the **\Registry** object, as shown in the following table.

USER-MODE HANDLE	CORRESPONDING OBJECT NAME
HKEY_LOCAL_MACHINE	**\Registry\Machine**
HKEY_USERS	**\Registry\User**
HKEY_CLASSES_ROOT	No kernel-mode equivalent
HKEY_CURRENT_USER	No simple kernel-mode equivalent, but see Registry Run-Time Library Routines

A driver can manipulate a registry-key object by performing the following steps:

1. Open a handle to the registry-key object. For more information, see Opening a Handle to a Registry-Key Object.
2. Perform the intended operations by calling the appropriate **Zw*Xxx*Key** routines. For information about how to do so, see Using a Handle to a Registry-Key Object.
3. Close the handle by calling **ZwClose**.

Opening a Handle to a Registry-Key Object

6/25/2019 • 2 minutes to read • Edit Online

To open a handle to a registry-key object, carry out the following two-step process:

1. Create an **OBJECT_ATTRIBUTES** structure, and initialize it by calling **InitializeObjectAttributes**. You specify the name of the key to manipulate as the *ObjectName* parameter to **InitializeObjectAttributes**.

   If you pass **NULL** as the *RootDirectory* parameter to **InitializeObjectAttributes**, *ObjectName* must be the full path of the registry key, beginning with **\Registry**. Otherwise, *RootDirectory* must be an open handle to a key, and *ObjectName* is the path that is relative to that key.

2. Open a handle to the key object by calling **ZwCreateKey** or **ZwOpenKey**, and pass the **OBJECT_ATTRIBUTES** structure to it. If the key does not already exist, **ZwCreateKey** will create the key, whereas **ZwOpenKey** will return STATUS_OBJECT_NAME_NOT_FOUND.

You pass a *DesiredAccess* parameter to **ZwCreateKey** or **ZwOpenKey** that contains the access rights you are requesting. You must specify the access rights that permit the operations your driver will perform. The following table lists the operations you can perform and the corresponding access rights to request.

OPERATION	REQUIRED ACCESS RIGHT
Get a registry-key value.	KEY_QUERY_VALUE or KEY_READ
Set a registry-key value.	KEY_SET_VALUE or KEY_WRITE
Loop through all of the subkeys of a key.	KEY_ENUMERATE_SUB_KEYS or KEY_READ
Create a subkey.	KEY_CREATE_SUB_KEY or KEY_WRITE
Delete a key.	DELETE

For more information about the available values for the *DesiredAccess* parameter, see **ZwCreateKey**.

You can also call **IoOpenDeviceRegistryKey** and **IoOpenDeviceInterfaceRegistryKey** to open handles to those registry keys that are device specific and device-interface specific, respectively. For more information, see Plug and Play Registry Routines.

Note For calls to **ZwCreateKey**, **ZwOpenKey**, **IoOpenDeviceRegistryKey**, and **IoOpenDeviceInterfaceRegistryKey**, the generic access rights, GENERIC_READ and GENERIC_WRITE, are equivalent in meaning to the key-specific access rights, KEY_READ and KEY_WRITE, respectively, and can be used as substitutes for these key-specific access rights.

Using a Handle to a Registry-Key Object

6/25/2019 • 2 minutes to read • Edit Online

The following table lists the operations that drivers can perform on an open key as well as the appropriate routines to call.

OPERATION	ROUTINE TO CALL
Examine the key's properties, such as its name or the number of its subkeys.	**ZwQueryKey**
Iterate through the key's subkeys, examining the properties of each one.	**ZwEnumerateKey**
Examine the properties of a key value, including the value's data.	**ZwQueryValueKey**
Iterate through a key's values, examining the properties of each one.	**ZwEnumerateValueKey**
Set the data for a value associated with a key.	**ZwSetValueKey**
Delete a key.	**ZwDeleteKey**
Delete a key value.	**ZwDeleteValueKey**

Once the driver has finished its manipulations, it must call **ZwClose** to close the handle—unless it has already called **ZwDeleteKey** to delete the key. (Once a key is deleted, all the open handles to it become invalid, so the driver must not close the handle in this case.)

The following code example illustrates how to open a handle for a key named **\Registry\Machine\Software***MyCompany**MyApp*, then retrieve key data and close the handle.

```
//
// Get the frame location from the registry key
// HKLM\SOFTWARE\MyCompany\MyApp.
// For example: "FrameLocation"="X:\\MyApp\\Frames"
//
HANDLE              handleRegKey = NULL;
for (int n = 0; n < 1; n++)
{
    NTSTATUS           status = NULL;
    UNICODE_STRING     RegistryKeyName;
    OBJECT_ATTRIBUTES  ObjectAttributes;

    RtlInitUnicodeString(&RegistryKeyName, L"\\Registry\\Machine\\Software\\MyCompany\\MyApp");
    InitializeObjectAttributes(&ObjectAttributes,
                               &RegistryKeyName,
                               OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                               NULL,    // handle
```

```
                                NULL);
    status = ZwOpenKey(&handleRegKey, KEY_READ, &ObjectAttributes);

    // If the driver cannot open the key, the driver cannot continue.
    // In this situation, the driver was probably set up incorrectly
    // and worst case, the driver cannot stream.
    if( NT_SUCCESS(status) == FALSE )
    {
        break;
    }
    // The driver obtained the registry key.
    PKEY_VALUE_FULL_INFORMATION  pKeyInfo = NULL;
    UNICODE_STRING               ValueName;
    ULONG                        ulKeyInfoSize = 0;
    ULONG                        ulKeyInfoSizeNeeded = 0;

    // The driver requires the following value.
    RtlInitUnicodeString(&ValueName, L"FrameLocation");

    // Determine the required size of keyInfo.
    status = ZwQueryValueKey( handleRegKey,
                              &ValueName,
                              KeyValueFullInformation,
                              pKeyInfo,
                              ulKeyInfoSize,
                              &ulKeyInfoSizeNeeded );

    // The driver expects one of the following errors.
    if( (status == STATUS_BUFFER_TOO_SMALL) || (status == STATUS_BUFFER_OVERFLOW) )
    {
        // Allocate the memory required for the key.
        ulKeyInfoSize = ulKeyInfoSizeNeeded;
        pKeyInfo = (PKEY_VALUE_FULL_INFORMATION) ExAllocatePoolWithTag( NonPagedPool, ulKeyInfoSizeNeeded,
g_ulTag);
        if( NULL == pKeyInfo )
        {
            break;
        }
        RtlZeroMemory( pKeyInfo, ulKeyInfoSize );

        // Get the key data.
        status = ZwQueryValueKey( handleRegKey,
                                  &ValueName,
                                  KeyValueFullInformation,
                                  pKeyInfo,
                                  ulKeyInfoSize,
                                  &ulKeyInfoSizeNeeded );
        if( (status != STATUS_SUCCESS) || (ulKeyInfoSizeNeeded != ulKeyInfoSize) || (NULL == pKeyInfo) )
        {
            break;
        }

        // Fill in the frame location if it has not been filled in already.
        if ( NULL == m_szwFramePath )
        {
            m_ulFramePathLength = pKeyInfo->DataLength;
            ULONG_PTR   pSrc = NULL;

            pSrc = (ULONG_PTR) ( (PBYTE) pKeyInfo + pKeyInfo->DataOffset);

            m_szwFramePath = (LPWSTR) ExAllocatePoolWithTag( NonPagedPool, m_ulFramePathLength, g_ulTag);
            if ( NULL == m_szwFramePath )
            {
                m_ulFramePathLength = 0;
                break;
            }

            // Copy the frame path.
            RtlCopyMemory(m_szwFramePath, (PVOID) pSrc, m_ulFramePathLength);
```

```
            RtlCopyMemory(m_szWrFrameFath, (PVOID) pSrc, m_dirFrameFathLength);
        }
        // The driver is done with the pKeyInfo.
        xFreePoolWithTag(pKeyInfo, g_ulTag);

    } // if( (status == STATUS_BUFFER_TOO_SMALL) || (status == STATUS_BUFFER_OVERFLOW) )
} // Get the Frame location from the registry key.

// All done with the registry.
if (NULL != handleRegKey)
{
    ZwClose(handleRegKey);
}
```

The system caches key changes in memory and writes them to disk every few seconds. To force a key change to disk, call **ZwFlushKey**.

To manipulate the registry through a simpler interface, drivers can also call the **Rtl*Xxx*Registry*Xxx*** routines. For more information, see Registry Run-Time Library Routines.

Registry Run-Time Library Routines

6/25/2019 • 2 minutes to read • Edit Online

To manipulate registry entries, drivers can call the **Rtl*Xxx*Registry*Xxx*** routines, which provide a simpler interface than the **Zw*Xxx*Key** routines. When doing so, the driver is not required to open and close handles; instead, the driver refers to keys by name.

You pass *RelativeTo* and *Path* parameters to each **Rtl*Xxx*Registry*Xxx*** routine. If *RelativeTo* is RTL_REGISTRY_ABSOLUTE, *Path* specifies the full path of the key, beginning with the **\Registry** root. If *RelativeTo* is RTL_REGISTRY_HANDLE, *Path* is actually an open handle. Additional RTL_REGISTRY_*XXX* values for *RelativeTo* specify the paths of common roots for the key; in these cases, *Path* specifies the path relative to that root. For example, RTL_REGISTRY_USER requires that *Path* be relative to the current user's registry settings. (This value is equivalent to specifying HKEY_CURRENT_USER in a user-mode application.) For a description of all the RTL_REGISTRY_*XXX* values, see **RtlCheckRegistryKey**.

The following table list the operations that drivers can perform by calling the **Rtl*Xxx*Registry*Xxx*** routines.

OPERATION	RTL*XXX*REGISTRY*XXX* ROUTINE TO CALL
Create a registry key	**RtlCreateRegistryKey**
Check whether a registry key exists	**RtlCheckRegistryKey**
Examine one or more registry-key values	**RtlQueryRegistryValues**
Write a registry-key value	**RtlWriteRegistryValue**
Delete a registry-key value	**RtlDeleteRegistryValue**

The following code example illustrates how to set *ValueName* for **\Registry\Machine\System***KeyName* to a ULONG value of 0xFF. Compare this example with the corresponding one in the Registry Key Object Routines section.

```
NTSTATUS status;
ULONG data = 0xFF;

status = RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,
                               (PWCSTR)L"\\Registry\\Machine\\System\\KeyName",
                               (PWCSTR)L"ValueName",
                               REG_DWORD,
                               &data,
                               sizeof(ULONG));
```

Although you write fewer lines of code when using the **Rtl*Xxx*Registry*Xxx*** routines instead of the **Zw*Xxx*Key** routines, the latter ones are necessary for performing certain operations. For example, no **Rtl*Xxx*Registry*Xxx*** routine exists that corresponds to **ZwEnumerateKey**.

If you perform multiple operations on the same key, the **Zw*Xxx*Key** routines are more efficient—you can use the

same open handle for each operation. In contrast, the **Rtl*Xxx*Registry*Xxx*** routines open and close a new handle for each operation.

Plug and Play Registry Routines

6/25/2019 • 2 minutes to read • Edit Online

The Plug and Play manager associates certain registry keys with a driver, its devices, and its device interface instances. Drivers can use these keys to store persistent properties associated with the driver, or with particular devices or device interface instances.

Drivers must never access these keys directly. Future versions of Windows may store the information at a different location in the registry, or outside the registry entirely. Drivers must not directly access any keys in the following trees:

- HKLM\SYSTEM\CurrentControlSet\Control\Class
- HKLM\SYSTEM\CurrentControlSet\Control\DeviceClasses
- HKLM\SYSTEM\CurrentControlSet\Enum
- HKLM\SYSTEM\CurrentControlSet\Hardware Profiles

Instead, drivers use the **IoOpenDeviceRegistryKey** and **IoOpenDeviceInterfaceRegistryKey** routines to access its PnP keys.

The PnP manager assigns one key for the driver, known as the driver's software key, and a key for each device, known as the device's hardware key. The **IoOpenDeviceRegistryKey** routine can be used to open either key. The value of the *DevInstKeyType* parameter determines which key to open. Specify PLUGPLAY_REGKEY_DRIVER to open a software key, or PLUGPLAY_REGKEY_DEVICE to a hardware key. The *DeviceObject* parameter specifies the device or driver. (The driver can also access its hardware and software keys relative to the current hardware profile, by ANDing PLUGPLAY_REGKEY_CURRENT_HWPROFILE to *DevInstKeyType*.)

IoOpenDeviceInterfaceRegistryKey opens the key associated with a particular device interface instance. The instance is identified by its name, which is a **UNICODE_STRING** returned by **IoGetDeviceInterfaces**, **IoGetDeviceInterfaceAlias**, or **IoRegisterDeviceInterface**. The string is passed as the *SymbolicLinkValue* parameter to **IoOpenDeviceInterfaceRegistryKey**.

These keys can also be set in an INF file, or by using the **SetupDi*Xxx*** routines. For more information, see Registry Keys for Drivers.

Both **IoOpenDeviceRegistryKey** and **IoOpenDeviceInterfaceRegistryKey** provide an open key handle, with access rights as specified by the *DesiredAccess* parameter. The driver subsequently uses the **Zw*Xxx*** registry routines, such as **ZwQueryValueKey** and **ZwSetValueKey**, to access and manipulate the key. After the driver is no longer using the handle, the driver closes the handle by calling **ZwClose**. For more information, see Using a Handle to a Registry-Key Object.

The following code sample demonstrates using **IoOpenDeviceRegistryKey** and **ZwSetValueKey** to set the data associated with the value named "Value" under the device's hardware key.

```
PDEVICE_OBJECT pDeviceObject; // A pointer to the PDO for the device.
HANDLE handle;
UNICODE_STRING ValueName;
ULONG Value = 109; // This is the value we're setting the key to.
NTSTATUS status;

RtlInitUnicodeString(&ValueName, L"Value");

status = IoOpenDeviceRegistryKey(pDeviceObject, PLUGPLAY_REGKEY_DEVICE, KEY_READ, &handle);

if (NTSUCCESS(status)) {
  status = ZwSetValueKey(handle, ValueName, 0, REG_DWORD, &Value, sizeof(ULONG));
  if (NTSUCCESS(status) {
    ZwClose(handle);
  } else {
    // Handle error.
  }
  // Handle error.
}
```

Note that access to a registry key can be restricted, so a call to **IoOpenDeviceRegistryKey** and **IoOpenDeviceInterfaceRegistryKey** should specify the minimum rights necessary for *DesiredAccess*. If the driver requests an access right that is not allowed, either routine returns STATUS_ACCESS_DENIED. In particular, drivers should not specify KEY_ALL_ACCESS.

Supporting Removable Media

12/5/2018 • 2 minutes to read • Edit Online

File systems and removable-media device drivers share the responsibility for ensuring that the correct media is mounted when a file is opened on a removable-media device and that the correct media remains mounted during operations that access the media. Any intermediate driver layered between a file system and a removable-media device driver also shares this responsibility.

Drivers that work with removable-media devices therefore should be capable of doing one or more of the following:

Responding to check-verify requests from the file system

Notifying the file system of possible media changes

Checking flags in the device object

Setting up IRPs in intermediate drivers

Responding to Check-Verify Requests from the File System

6/25/2019 • 2 minutes to read • Edit Online

At its discretion, the file system can send an IRP to the device driver's Dispatch entry point for **IRP_MJ_DEVICE_CONTROL** requests with **Parameters.DeviceIoControl.IoControlCode** in the I/O stack location set to the following:

IOCTL_*XXX*_CHECK_VERIFY
where *XXX* is the type of device, such as DISK, TAPE, or CDROM.

The type DISK includes both unpartitionable (floppy) and partitionable removable-media devices.

If the underlying device driver determines that the media has not changed, the driver should complete the IRP, returning the **IoStatus** block with the following values:

Status	Set to STATUS_SUCCESS
Information	Set to zero

In addition, if the device type is DISK or CDROM and the caller specified an output buffer, the driver returns the media change count in the buffer at **Irp->AssociatedIrp.SystemBuffer** and sets **Irp->IoStatus.Information** to **sizeof**(ULONG). By returning this count, the driver gives the caller an opportunity to determine whether the media has changed from its perspective.

If the underlying device driver determines that the media has changed, it takes a different action depending on whether the volume is mounted. If the volume is mounted (the VPB_MOUNTED flag is set in the VPB), the driver should do the following:

1. Set the **Flags** in the **DeviceObject** by ORing **Flags** with DO_VERIFY_VOLUME.
2. Set the **IoStatus** block in the IRP to the following:
   - **Status** set to STATUS_VERIFY_REQUIRED
   - **Information** set to zero
3. Call **IoCompleteRequest** with the input IRP.

If the volume is not mounted, the driver must not set the DO_VERIFY_VOLUME bit. The driver should set **IoStatus.Status** to STATUS_IO_DEVICE_ERROR, set **IoStatus.Information** to zero, and call **IoCompleteRequest** with the IRP.

Notifying the File System of Possible Media Changes

6/25/2019 • 2 minutes to read • Edit Online

A removable-media device driver must ensure that the media is not changed for the device represented by the **DeviceObject** (input to every driver routine that is sent an IRP) whenever the driver processes an IRP that requests a transfer to/from the media or a device I/O control operation that affects the media. The best possible time to check for changed media is just after the transition from a no-media-present state to a media-present state if the physical device always notifies the driver about these state changes.

If its physical device indicates that the state of the media might have changed before the driver begins an I/O operation or during an operation, the driver must do the following:

1. Ensure that the volume is mounted by checking the VPB_MOUNTED flag in the *VPB*. (If the volume is not mounted, the driver must not set the DO_VERIFY_VOLUME bit. The driver should set **IoStatus.Status** to STATUS_IO_DEVICE_ERROR, set **IoStatus.Information** to zero, and call **IoCompleteRequest** with the IRP.)

2. Set the **Flags** in the **DeviceObject** by ORing **Flags** with DO_VERIFY_VOLUME.

3. Set the **IoStatus** block in the IRP to the following:

   - **Status** set to STATUS_VERIFY_REQUIRED
   - **Information** set to zero

4. Before completing any IRP with an **IoStatus** block in which the **Status** field is not set to STATUS_SUCCESS, the driver must call **IoIsErrorUserInduced**, which returns a Boolean **TRUE** for any of the following **Status** values:

   - STATUS_VERIFY_REQUIRED
   - STATUS_NO_MEDIA_IN_DEVICE
   - STATUS_WRONG_VOLUME
   - STATUS_UNRECOGNIZED_MEDIA
   - STATUS_MEDIA_WRITE_PROTECTED
   - STATUS_IO_TIMEOUT
   - STATUS_DEVICE_NOT_READY

   If **IoIsErrorUserInduced** returns **TRUE**, the driver must call **IoSetHardErrorOrVerifyDevice** so the FSD can open a dialog box to the user, who can then choose to supply the correct media, retry the original request, or cancel the requested operation.

Checking Flags in the Device Object

6/25/2019 • 2 minutes to read • Edit Online

For each IRP requesting an I/O operation to/from removable media, a removable-media device driver must determine whether DO_VERIFY_VOLUME is already set in its **DeviceObject->Flags**. If this value is set, the driver must do the following:

- For **IRP_MJ_READ**, **IRP_MJ_WRITE**, and **IRP_MJ_DEVICE_CONTROL** requests, check whether SL_OVERRIDE_VERIFY_VOLUME is set in the **Flags** member of the driver's **IO_STACK_LOCATION** structure. If it is, continue the requested operation.

  Device control requests that return information about the logical structure of the underlying media have SL_OVERRIDE_VERIFY_VOLUME set in the I/O stack location's **Flags** member when an IFS mounts or remounts a removable-media volume.

- Otherwise, the driver must refuse to carry out I/O requests for the corresponding drive, device, or partition while DO_VERIFY_VOLUME is set in its **DeviceObject->Flags**. A removable media driver must fail IRPs sent to the corresponding device as described in the preceding subsection, repeating both Steps 3 and 4 for each IRP until the FSD clears DO_VERIFY_VOLUME in the removable-media driver's **DeviceObject->Flags**.

If a removable-media device driver does not fail IRPs when DO_VERIFY_VOLUME is set and SL_OVERRIDE_VERIFY_VOLUME is not set for the preceding transfer requests, the file system can neither maintain the integrity of cached file data nor cause the user to be prompted to remount the media that holds an open file.

Setting up IRPs in Intermediate Drivers

6/25/2019 • 2 minutes to read • Edit Online

Any intermediate driver layered between a file system driver and a removable-media device driver must set up the next-lower-level driver's I/O stack location in IRPs. From incoming **IRP_MJ_READ**, **IRP_MJ_WRITE**, and **IRP_MJ_DEVICE_CONTROL** requests, the intermediate driver must copy its own I/O stack location **Flags** into the next-lower-level driver's I/O stack location when it sets up the I/O stack location for the lower driver.

If the intermediate driver allocates new IRPs for lower-level removable-media drivers, it must set up those IRPs as follows:

- For transfer requests, it must set up the thread context in each driver-allocated IRP from the value at **Tail.Overlay.Thread** in the original IRP.
- For **IRP_MJ_READ**, **IRP_MJ_WRITE**, and **IRP_MJ_DEVICE_CONTROL** requests, it must copy the I/O stack location **Flags** from the original IRP to each driver-allocated IRP.

Otherwise, the file system can neither maintain the integrity of cached file data nor cause the user to be prompted to remount the media that holds an open file.

Creating Export Drivers

6/25/2019 • 3 minutes to read • Edit Online

Microsoft Windows drivers are typically defined as a pair of components, such as a port/miniport driver pair, or a class/miniclass driver pair. Typically, Microsoft provides a hardware-independent class or port driver, and a vendor supplies a hardware-dependent miniclass or miniport driver.

Kernel-mode export drivers are especially well suited for implementing the part of a driver pair that is independent of underlying stack and hardware characteristics, because an export driver is a kernel-mode DLL that can be loaded by a variety of other hardware-specific or device-stack-specific components. Microsoft ships several drivers together with the Windows operating system that fall into this category. For example, the SCSI port driver, the tape class driver, the IDE controller driver are all system-supplied export drivers that are loaded by other drivers.

An export driver is missing many of the characteristics of a complete kernel-mode driver. An export driver does not have a dispatch table, it does not have a place in the driver stack, and it does not have an entry in the service control manager's database that defines it as a system service. An export driver does have a **DriverEntry** routine, but its **DriverEntry** routine is never called. (The routine is only a stub to satisfy the requirements of the build scripts.)

Note that, while an export driver does not have a dispatch table, it can supply dispatch routines to a standard driver. The standard driver inserts the dispatch routines into its own dispatch table.

Standard drivers can also function as export drivers. For a driver to function in both ways, it must be built as an export driver and loaded as a regular driver.

Building an Export Driver

To build a driver as an export driver you must define several Build utility macros in the driver's Sources file.

First, you must assign the appropriate value to the **TARGETTYPE** macro, as follows:

```
TARGETTYPE=EXPORT_DRIVER
```

You must also specify a module-definition (.def) file using the **DLLDEF** macro. For example:

```
DLLDEF="c:\project\driver.def"
```

The module-definition file provides the compiler and linker with a list of exported routines along with other information. For more information about module-definition files, see the Microsoft Visual C++ documentation.

Many of the Build utility macros employed in building a user-mode DLL cannot be used when building a kernel-mode DLL.

For instance, the entry point for a kernel-mode DLL is always **DllInitialize**. The system calls a kernel-mode DLL's DllInitialize routine immediately after the DLL is loaded. Export drivers must provide **DllInitialize** routines. You can use the **DllInitialize** routine to acquire or initialize resources required by other routines in the DLL.

You cannot specify the entry point using the **DLLENTRY** macro.

```
NTSTATUS DllInitialize(
  _In_ PUNICODE_STRING RegistryPath
);
```

RegistryPath is a pointer to a counted Unicode string specifying the path to the DLL's registry key, **HKEY_LOCAL_MACHINE\CurrentControlSet\Services\DllName**. DLL routines can use this key to store DLL-specific information. The buffer pointed to by RegistryPath is freed once **DllInitialize** exits. Therefore, if the DLL makes use of the key, **DllInitialize** must duplicate the key name.

The build process generates an export library with a .lib extension, and an export driver with a .sys extension.

Importing Functions from an Export Driver

To import functions that are exported by an export driver, you should declare the functions using the DECLSPEC_IMPORT macro, which is defined in Ntdef.h. For example:

```
DECLSPEC_IMPORT int LoadPrinterDriver (int arg1);
```

This macro resolves to a **__declspec**(dllimport) storage class declaration on those platforms where required and to nothing on those platforms where not required.

In the export driver, the function to be exported should be declared with the DECLSPEC_EXPORT macro. This macro resolves to a **__declspec**(dllexport) storage class declaration on those platforms where required and to nothing on those platforms where not required. If an export driver supplies a dispatch routine to a standard driver, that routine does not have to be exported.

Loading and Unloading an Export Driver

Export drivers must be installed in the %Windir%\System32\Drivers directory. Starting with Windows 2000, the operating system keeps a reference count that indicates the number of times that the export driver's functions have been imported by other drivers. The system decrements this count whenever one of the importing drivers unloads. When the reference count falls to zero, the system unloads the export driver. However, the export driver must contain the standard entry point and unload routines, **DllInitialize** and **DllUnload**, or the operating system will not activate this reference count mechanism.

The system calls a kernel-mode DLL's DllUnload routine when it unloads the DLL.

```
NTSTATUS DllUnload(void);
```

Export drivers must provide DllUnload routines. You can use the DllUnload routine to release any resources used by the routines in the DLL.

Creating Reliable Kernel-Mode Drivers

6/25/2019 • 3 minutes to read • Edit Online

Drivers make up a significant percentage of the total code that executes in kernel mode. A kernel-mode driver is, in effect, a component of the operating system. Therefore, drivers that are reliable and secure contribute significantly to the overall trustworthiness of the operating system. To create a reliable kernel-mode driver, follow these guidelines:

- Secure device objects properly.

  User access to a system's drivers and devices is controlled by security descriptors that the system assigns to device objects. Most often, the system sets device security parameters when a device is installed. For more information, see Creating Secure Device Installations. Sometimes it is appropriate for a driver to play a part in controlling access to its device. For more information, see Securing Device Objects.

- Validate device objects properly.

  If a driver creates multiple types of device objects, it must check which type it receives in each IRP. For more information, see Failure to Validate Device Objects.

- Use "safe string" functions.

  When manipulating strings, a driver should use safe string functions instead of the string functions that are supplied with C/C++ language runtime libraries. For more information, see Using Safe String Functions.

- Validate object handles.

  Drivers that receive object handles as input must verify that the handles are valid, are accessible, and are of the type expected. For more information about using object handles, see the following topics:

  Object Management

  Failure to Validate Object Handles

- Support multiprocessors properly.

  Never assume that your driver will run only on single-processor systems. For information about programming techniques that you can use to ensure that your driver will function properly on multiprocessor systems, see the following topics:

  Synchronization Techniques

  Errors in a Multiprocessor Environment

- Handle driver state properly.

  It is important to always verify that your driver is in the state you assume it to be in. For example, if the driver receives an IRP, is it already servicing an IRP of the same type? If the driver does not check for this situation, the first IRP could be lost. For more information, see Failure to Check a Driver's State.

- Validate IRP input values.

  It is essential, from both a reliability and a security perspective, to validate all values that are associated with an IRP, such as buffer addresses and lengths. The following topics provide information about validating IRP input values:

  DispatchReadWrite Using Buffered I/O

Errors in Buffered I/O

DispatchReadWrite Using Direct I/O

Errors in Direct I/O

Security Issues for I/O Control Codes

Errors in Referencing User-Space Addresses

- Handle the I/O stack properly.

  When passing IRPs down the driver stack, it is important for drivers to call **IoSkipCurrentIrpStackLocation** or **IoCopyCurrentIrpStackLocationToNext** to set up the next driver's I/O stack location. Do not write code that directly copies one I/O stack location to the next.

- Handle IRP completion operations properly.

  A driver must never complete an IRP with a status value of STATUS_SUCCESS unless it actually supports and processes the IRP. For information about the correct ways to handle IRP completion operations, see Completing IRPs.

- Handle IRP cancellation operations properly.

  Cancel operations can be difficult to code properly because they typically execute asynchronously. Problems in the code that handles cancel operations can go unnoticed for a long time, because this code is typically not executed frequently in a running system.

  Be sure to read and understand all of the information supplied under Canceling IRPs. Pay special attention to Synchronizing IRP Cancellation and Points to Consider When Canceling IRPs.

  One way to avoid the synchronization problems that are associated with cancel operations is to implement a cancel-safe IRP queue. A cancel-safe IRP queue is a driver-managed queue that was introduced for Windows XP and later operating system versions, but is also backward-compatible to earlier versions.

- Handle IRP cleanup and close operations properly.

  Be sure that you understand the difference between **IRP_MJ_CLEANUP** and **IRP_MJ_CLOSE** requests. Cleanup requests arrive after an application closes all handles on a file object, but sometimes before all I/O requests have completed. Close requests arrive after all I/O requests for the file object have been completed or canceled. For more information, see the following topics:

  DispatchCreate, DispatchClose, and DispatchCreateClose Routines

  DispatchCleanup Routines

  Errors in Handling Cleanup and Close Operations

For more information about handling IRPs correctly, see Additional Errors in Handling IRPs.

Using Driver Verifier

Driver Verifier is the most important tool you can use to ensure the reliability of your driver. Driver Verifier can check for a variety of common driver problems, including some of those discussed in this section. However, use of Driver Verifier does not replace careful, thoughtful software design.

Failure to Validate Device Objects

6/25/2019 • 2 minutes to read • Edit Online

Many drivers create more than one kind of device object by calling **IoCreateDevice**. Some drivers create control device objects in their **DriverEntry** routines to allow applications to communicate with the driver, even before the driver creates an FDO. For example, file system drivers create device objects to handle file system notifications when they register themselves as file systems with **IoRegisterFileSystem**.

A driver should be ready for **IRP_MJ_CREATE** requests for any device object it creates. After completing the request with a success status, the driver should expect to receive any user-accessible I/O requests on the created file object. Consequently, any driver that creates more than one device object must check which device object each I/O request specifies.

For example, suppose a driver creates overall control device objects in **DriverEntry**, and then creates another set of device objects in its *AddDevice* routine. Suppose the *AddDevice* routine initializes the device extension with information about lower-level drivers, but the control device objects do not contain this information. In this case, all dispatch routines must be careful to check each device object that they receive. Otherwise, the driver might crash when trying to use device extension information.

Failure to Validate Object Handles

6/25/2019 • 2 minutes to read • Edit Online

Some drivers must manipulate objects passed to them by callers or must handle two file objects at the same time. For example, a modem driver might receive a handle to an event object, or a network driver might receive handles to two different file objects. The driver must validate these handles. Because they are passed by a caller, and not through the I/O manager, the I/O manager cannot perform any validation checks.

For example, in the following code snippet, the driver has been passed the handle **AscInfo->AddressHandle**, but has not validated it before calling **ObReferenceObjectByHandle**:

```
//
// This handle is embedded in a buffered request.
//
status = ObReferenceObjectByHandle(
                    AscInfo->AddressHandle,
                    0,
                    NULL,
                    KernelMode,
                    &fileObject,
                    NULL);

if (NT_SUCCESS(status)) {
    if ( (fileObject->DeviceObject == DeviceObject) &&
         (fileObject->FsContext2 == TRANSPORT_SOCK) ) {
```

Although the call to **ObReferenceObjectByHandle** succeeds, the code fails to ensure that the returned pointer references a file object; it trusts the caller to pass in the correct information.

Even if all the parameters for the call to **ObReferenceObjectByHandle** are correct, and the call succeeds, a driver can still get unexpected results if the file object is not intended for its driver. In the following code fragment, the driver assumes that a successful call returns a pointer to the file object it expected:

```
status = ObReferenceObjectByHandle (
                           AcpInfo->Handle,
                           0L,
                           DesiredAccess,
                           *IoFileObjectType,
                           Irp->RequestorMode,
                           (PVOID *)&AcpEndpointFileObject,
                           NULL);

if ( !NT_SUCCESS(status) ) {
   goto complete;
}
AcpEndpoint = AcpEndpointFileObject->FsContext;

if ( AcpEndpoint->Type != BlockTypeEndpoint )
```

Although **ObReferenceObjectByHandle** returns a pointer to a file object, the driver has no guarantee that the pointer references the file object it expected. In this case, the driver should validate the pointer before accessing the driver-specific data at **AcpEndpointFileObject->FsContext**.

To avoid such problems, a driver should check for valid data, as follows:

- Check the object type to make sure it is what the driver expects.

- Ensure that the requested access is appropriate for the object type and required tasks. If your driver performs a fast file copy, for instance, make sure the handle has read access.
- Be sure to specify the correct access mode (**UserMode** or **KernelMode**) and that the access mode is compatible with the access requested.
- If the driver expects a handle to a file object that the driver itself created, validate the handle against the device object or driver. However, be careful not to break filters that send I/O requests for strange devices.
- If your driver supports multiple kinds of file objects (such as the control channels, address objects, and connections of TDI drivers or Volume, Directory, and File objects of file systems), make sure you have a way to differentiate them.

Errors in a Multiprocessor Environment

12/5/2018 • 2 minutes to read • Edit Online

On the NT-based operating system, drivers are multithreaded; they can receive multiple I/O requests from different threads at the same time. In designing a driver, you must assume that it will be run on an SMP system and take the appropriate measures to ensure data integrity.

Specifically, whenever a driver changes global or file object data, it must use a lock or an interlocked sequence to prevent race conditions.

Encountering a race condition when referencing global or file object-specific data

In the following code snippet, a race condition could occur when the driver accesses the global data at **Data.LpcInfo**:

```
PLPC_INFO pLpcInfo = &Data.LpcInfo; //Pointer to global data
...
...
// This saved pointer may be overwritten by another thread.
pLpcInfo->LpcPortName.Buffer = ExAllocatePool(
                                  PagedPool,
                                  arg->PortName.Length);
```

Multiple threads entering this code as a result of an IOCTL call could cause a memory leak as the pointer is overwritten. To avoid this problem, the driver should use the **ExInterlocked*Xxx*** routines or some type of lock when it changes the global data. The driver's requirements determine the acceptable types of locks. For further information, see Spin Locks, Kernel Dispatcher Objects, and **ExAcquireResourceSharedLite**.

The following example attempts to reallocate a file-specific buffer (**Endpoint->LocalAddress**) to hold the endpoint address:

```
Endpoint = FileObject->FsContext;

 if ( Endpoint->LocalAddress != NULL &&
      Endpoint->LocalAddressLength <
                ListenEndpoint->LocalAddressLength ) {

   FREE_POOL (Endpoint->LocalAddress,
              LOCAL_ADDRESS_POOL_TAG
              );
   Endpoint->LocalAddress  = NULL;
}

 if ( Endpoint->LocalAddress == NULL ) {
    Endpoint->LocalAddress =
         ALLOCATE_POOL (NonPagedPool,
                        ListenEndpoint->LocalAddressLength,
                        LOCAL_ADDRESS_POOL_TAG);
}
```

In this example, a race condition could occur with accesses to the file object. Because the driver does not hold any locks, two requests for the same file object can enter this function. The result might be references to freed memory, multiple attempts to free the same memory, or memory leaks. To avoid these errors, the two **if** statements should be enclosed in a spin lock.

Failure to Check a Driver's State

6/25/2019 • 2 minutes to read • Edit Online

In the following example, the driver uses the **ASSERT** macro to check for the correct device state in the checked build, but does not check device state in the free build:

```
case IOCTL_WAIT_FOR_EVENT:

   ASSERT((!Extension->WaitEventIrp));
   Extension->WaitEventIrp = Irp;
   IoMarkIrpPending(Irp);
   status = STATUS_PENDING;
```

In the checked build, if the driver already holds the IRP pending, the system will assert. In the free build, however, the driver does not check for this error. Two calls to the same IOCTL cause the driver to lose track of an IRP.

On a multiprocessor system, this code fragment might cause additional problems. Assume that on entry this routine has ownership of (the right to manipulate) this IRP. When the routine saves the **Irp** pointer in the global structure at **Extension->WaitEventIrp**, another thread can get the IRP address from that global structure and perform operations on the IRP. To prevent this problem, the driver should mark the IRP pending before it saves the IRP and should include both the call to **IoMarkIrpPending** and the assignment in an interlocked sequence. A *Cancel* routine for the IRP might also be necessary.

Failure to Check the Size of Buffers

12/5/2018 • 2 minutes to read • Edit Online

When handling IOCTLs and FSCTLs that implement buffered I/O, a driver should always check the sizes of the input and output buffers to ensure that the buffers can hold all the requested data. If the request specifies FILE_ANY_ACCESS, as most driver IOCTLs and FSCTLs do, any caller that has a handle to the device has access to buffered IOCTL or FSCTL requests for that device, and could read or write data beyond the end of the buffer.

Input Buffer Size

For example, assume that the following code appears in a routine that is called from a *Dispatch* routine, and that the driver has not validated the buffer sizes passed in the IRP:

```
switch (ControlCode)
   ...
   ...
   case IOCTL_NEW_ADDRESS:{
      tNEW_ADDRESS *pNewAddress =
         pIrp->AssociatedIrp.SystemBuffer;

      pDeviceContext->Addr = RtlUlongByteSwap (pNewAddress->Address);
```

The example does not check the buffer sizes before the assignment statement (highlighted). As a result, the **pNewAddress->Address** reference in the next line can fault if the input buffer is not big enough to contain a tNEW_ADDRESS structure.

The following code checks the buffer sizes, avoiding the potential problem:

```
case IOCTL_NEW_ADDRESS: {
   tNEW_ADDRESS *pNewAddress =
      pIrp->AssociatedIrp.SystemBuffer;

   if (pIrpSp->Parameters.DeviceIoControl.InputBufferLength >=
          sizeof(tNEW_ADDRESS)) {
      pDeviceContext->Addr = RtlUlongByteSwap (pNewAddress->Address);
```

Code to handle other buffered I/O, such as WMI requests that use variable size buffers, can have similar errors.

Output Buffer Size

Output buffer problems are similar to input buffer problems. They can easily corrupt pool, and user-mode callers may be unaware that any error has occurred.

In the following example, the driver fails to check the size of the **SystemBuffer**:

```
case IOCTL_GET_INFO: {

    Info = Irp->AssociatedIrp.SystemBuffer;

    Info->NumIF = NumIF;
    ...
    ...
    Irp->IoStatus.Information =
          NumIF*sizeof(GET_INFO_ITEM)+sizeof(ULONG);
    Irp->IoStatus.Status = ntStatus;
}
```

Assuming that the **NumIF** field of the system buffer specifies the number of input items, this example can set **IoStatus.Information** to a value larger than the output buffer and thus return too much information to the user-mode code. If an application is improperly coded, and calls with too small an output buffer, the preceding code could corrupt the pool by writing beyond the end of the system buffer.

Remember that the I/O manager assumes that the value in the **Information** field is valid. If a caller passes in a valid kernel-mode address for the output buffer and a size of zero bytes, serious problems can occur if the driver does not check the output buffer size and thus find the error.

Failure to Initialize Output Buffers

6/25/2019 • 2 minutes to read • Edit Online

Drivers should initialize all output buffers with zeros before returning them to the caller. Failing to initialize a buffer can result in garbage data in any uninitialized bytes.

In the following example, a driver returns garbage in unused bytes.

```
case IOCTL_GET_NAME: {
   ...
   ...
   outputBufferLength =
      ioStack->Parameters.DeviceIoControl.OutputBufferLength;
   outputBuffer = (PGET_NAME) Irp->AssociatedIrp.SystemBuffer;

   if (outputBufferLength >= sizeof(GET_NAME)) {
      length = outputBufferLength - sizeof(GET_NAME);

      ntStatus = IoGetDeviceProperty(
                     DeviceExtension->PhysicalDeviceObject,
                     DevicePropertyDriverKeyName,
                     length,
                     outputBuffer->DriverKeyName,
                     &length);

      outputBuffer->ActualLength =
                     length + sizeof(GET_NAME);

      Irp->IoStatus.Information = outputBufferLength;

   } else {
      ntStatus = STATUS_BUFFER_TOO_SMALL;
   }
```

Setting **IoStatus.Information** to the output buffer size causes the whole output buffer to be returned to the caller. The I/O manager does not initialize the data beyond the size of the input buffer—the input and output buffers overlap for a buffered request. Because the system support routine **IoGetDeviceProperty** does not write the entire buffer, this IOCTL returns uninitialized data to the caller.

Some drivers use the **Information** field to return codes that provide extra details about I/O requests. Before doing so, such drivers should check the IRP flags to ensure that IRP_INPUT_OPERATION is not set. When this flag is not set, the IOCTL or FSCTL does not have an output buffer, so the **Information** field need not supply a buffer size. In this case. the driver can safely use the **Information** field to return its own code.

Failure to Validate Variable-Length Buffers

12/5/2018 • 2 minutes to read • Edit Online

Drivers often accept input buffers with fixed headers and trailing variable length data, as in the following example:

```
typedef struct _WAIT_FOR_BUFFER {
   LARGE_INTEGER Timeout;
   ULONG NameLength;
   BOOLEAN TimeoutSpecified;
   WCHAR Name[1];
} WAIT_FOR_BUFFER, *PWAIT_FOR_BUFFER;

if (InputBufferLength < sizeof(WAIT_FOR_BUFFER)) {
   IoCompleteRequest( Irp, STATUS_INVALID_PARAMETER );
   return( STATUS_INVALID_PARAMETER );
}

WaitBuffer = Irp->AssociatedIrp.SystemBuffer;

if (FIELD_OFFSET(WAIT_FOR_BUFFER, Name[0]) +
       WaitBuffer->NameLength > InputBufferLength) {
    IoCompleteRequest( Irp, STATUS_INVALID_PARAMETER );
    return( STATUS_INVALID_PARAMETER );
}
```

If **WaitBuffer->NameLength** is a very large ULONG value, adding it to the offset could cause an integer overflow. Instead, a driver should subtract the offset from the **InputBufferLength**, and compare the result with **WaitBuffer->NameLength**, as in the following example:

```
if (InputBufferLength < sizeof(WAIT_FOR_BUFFER)) {
   IoCompleteRequest( Irp, STATUS_INVALID_PARAMETER );
   Return( STATUS_INVALID_PARAMETER );
}

WaitBuffer = Irp->AssociatedIrp.SystemBuffer;

if ((InputBufferLength -
      FIELD_OFFSET(WAIT_FOR_BUFFER, Name[0])  >
      WaitBuffer->NameLength) {
   IoCompleteRequest( Irp, STATUS_INVALID_PARAMETER );
   return( STATUS_INVALID_PARAMETER );
}
```

The subtraction above cannot underflow because the first **if** statement ensures that the **InputBufferLength** is greater than or equal to the size of **WAIT_FOR_BUFFER**.

The following shows a more complicated overflow problem:

```
case IOCTL_SET_VALUE:
   dwSize = sizeof(SET_VALUE);

   if (inputBufferLength < dwSize) {
      ntStatus = STATUS_BUFFER_TOO_SMALL;
      break;
   }

   dwSize = FIELD_OFFSET(SET_VALUE, pInfo[0]) +
               pSetValue->NumEntries * sizeof(SET_VALUE_INFO);

   if (inputBufferLength < dwSize) {
      ntStatus = STATUS_BUFFER_TOO_SMALL;
      break;
   }
```

In this example, an integer overflow can occur during multiplication. If the size of the **SET_VALUE_INFO** structure is a multiple of 2, a **NumEntries** value such as 0x80000000 results in an overflow, when the bits are shifted left during multiplication. However, the buffer size will nevertheless pass the validation test, because the overflow causes **dwSize** to appear quite small. To avoid this problem, subtract the lengths as in the previous example, divide by **sizeof**(**SET_VALUE_INFO**), and compare the result with **NumEntries**.

Errors in Direct I/O

6/25/2019 • 2 minutes to read • Edit Online

The most common direct I/O problem is failing to handle zero-length buffers correctly. Because the I/O manager does not create MDLs for zero-length transfers, a zero-length buffer results in a **NULL** value at **Irp->MdlAddress**.

To map the address space, drivers should use **MmGetSystemAddressForMdlSafe**, which returns **NULL** if mapping fails, as it will if a driver passes a **NULL MdlAddress**. Drivers should always check for a **NULL** return before attempting to use the returned address.

Direct I/O involves double-mapping the user's address space to a system address buffer, so that two different virtual addresses have the same physical address. Double-mapping has the following consequences, which can sometimes cause problems for drivers:

- The offset into the virtual page of the user's address becomes the offset into the system page.

  Access beyond the end of these system buffers may go unnoticed for long periods of time depending on the page granularity of the mapping. Unless a caller's buffer is allocated near the end of a page, data written beyond the end of the buffer will nevertheless appear in the buffer, and the caller will be unaware that any error has occurred. If the end of the buffer coincides with the end of a page, the system virtual addresses beyond the end could point to anything or could be invalid. Such problems can be extremely difficult to find.

- If the calling process has another thread that modifies the user's mapping of the memory, the contents of the system buffer will change when the user's memory mapping changes.

  In this situation, using the system buffer to store scratch data can cause problems. Two fetches from the same memory location might yield different values.

  The following code snippet receives a string in a direct I/O request, then tries to convert that string to uppercase characters:

  ```
  PWCHAR  PortName = NULL;

  PortName = (PWCHAR)MmGetSystemAddressForMdlSafe(irp->MdlAddress, NormalPagePriority);

  //
  // Null-terminate the PortName so that RtlInitUnicodeString will not
  // be invalid.
  //
  PortName[Size / sizeof(WCHAR) - 1] = UNICODE_NULL;

  RtlInitUnicodeString(&AdapterName, PortName);
  ```

  Because the buffer might not be correctly formed, the code attempts to force a Unicode **NULL** as the last buffer character. However, if the underlying physical memory is doubly mapped to both a user- and a kernel-mode address, another thread in the process can overwrite the buffer as soon as this write operation completes.

  Conversely, if the **NULL** is not present, then the call to **RtlInitUnicodeString** can exceed the range of the buffer and possibly cause a bug check if it falls outside the system mapping.

If a driver creates and maps its own MDL, it should ensure that it accesses the MDL only with the method for which it has probed. That is, when the driver calls **MmProbeAndLockPages**, it specifies an access method (**IoReadAccess**, **IoWriteAccess**, or **IoModifyAccess**). If the driver specifies **IoReadAccess**, it must not later

attempt to write to the system buffer made available by **MmGetSystemAddressForMdl** or **MmGetSystemAddressForMdlSafe**.

Errors in Referencing User-Space Addresses

6/25/2019 • 2 minutes to read • Edit Online

Any driver, whether supporting IRPs or fast I/O operations, should validate any address in user space before trying to use it. The I/O manager does not validate such addresses, nor does it validate pointers that are embedded in buffers passed to drivers.

Failure to Validate Addresses Passed in METHOD_NEITHER IOCTLs and FSCTLs

The I/O manager does no validation whatsoever for METHOD_NEITHER IOCTLs and FSCTLs. To ensure that user-space addresses are valid, the driver must use the **ProbeForRead** and **ProbeForWrite** routines, enclosing all buffer references in **try/except** blocks.

In the following example, the driver assumes that the value passed in the **Type3InputBuffer** represents a valid address.

```
case IOCTL_GET_HANDLER:
{
   PULONG EntryPoint;

   EntryPoint =
      IrpSp->Parameters.DeviceIoControl.Type3InputBuffer;
   *EntryPoint = (ULONG)DriverEntryPoint;
   ...
}
```

The following code avoids this problem:

```
case IOCTL_GET_HANDLER:
{
   PULONG_PTR EntryPoint;

   EntryPoint =
      IrpSp->Parameters.DeviceIoControl.Type3InputBuffer;

   try
   {
      if (Irp->RequestorMode != KernelMode)
      {
         ProbeForWrite(EntryPoint,
                       sizeof(ULONG_PTR),
                       TYPE_ALIGNMENT(ULONG_PTR));
      }
      *EntryPoint = (ULONG_PTR)DriverEntryPoint;
   }
   except(EXCEPTION_EXECUTE_HANDLER)
   {
     ...
   }
   ...
}
```

Note also that the correct code casts **DriverEntryPoint** to a ULONG_PTR, instead of a ULONG. This change allows for use in a 64-bit Windows environment.

Failure to validate pointers embedded in buffered I/O requests

Often drivers embed pointers within buffered requests, as in the following example:

```
struct ret_buf
{
   void  *arg;  // Pointer embedded in request
   int  rval;
};

pBuf = Irp->AssociatedIrp.SystemBuffer;
...
arg = pBuf->arg;  // Fetch the embedded pointer
...
// If the arg pointer is not valid, the following
// statement can corrupt the system:
RtlMoveMemory(arg, &info, sizeof(info));
```

In this example, the driver should validate the embedded pointer by using the **Probe*Xxx*** routines enclosed in a **try/except** block in the same way as for the METHOD_NEITHER IOCTLs described earlier. Although embedding a pointer allows a driver to return extra information, a driver can more efficiently achieve the same result by using a relative offset or a variable length buffer.

For more information about using **try/except** blocks to handle invalid addresses, see Handling Exceptions.

Errors in Handling Cleanup and Close Operations

12/7/2018 • 2 minutes to read • Edit Online

Some drivers fail to distinguish the tasks required in *DispatchCleanup* and *DispatchClose* routines. The I/O manager calls a driver's *DispatchCleanup* routine when the last handle to a file object is closed. The *DispatchClose* routine is called when the last reference is released from the file object. A driver should not attempt to free resources in its *DispatchCleanup* routine that are attached to a file object or might be used by other *Dispatch*Xxx routines.

When calling dispatch routines, the I/O manager holds a reference to the file object for normal I/O calls. As a result, a driver can receive I/O requests for a file object after its *DispatchCleanup* routine has been called but before its *DispatchClose* routine is called. For example, a user-mode caller might close the file handle while an I/O manager request is in progress from another thread. If the driver has deleted or freed necessary resources before the I/O manager calls its *DispatchClose* routine, invalid pointer references and other problems could occur.

Additional Errors in Handling IRPs

6/25/2019 • 2 minutes to read • Edit Online

The following are additional errors that drivers sometimes make when handling IRPs.

Lost or double-completed IRPs

These problems, along with missing calls to I/O manager routines such as **IoStartNextPacket**, often occur in error-handling paths. Quick reviews of driver paths can find such problems.

Converging public IOCTL and private IOCTL paths

As a general rule, drivers should contain separate execution paths for public and private IOCTLs (or FSCTLs). A driver cannot determine whether an IOCTL or FSCTL request originates in kernel mode or user mode by looking at the control code. Consequently, handling both public and private codes in the same execution path (or performing minimal validation and then calling the same routines) can open a driver to security breaches. If a private IOCTL or FSCTL is privileged, then unprivileged users who know the control codes might be able to gain access to it. Therefore, if your driver supports private IOCTL or FSCTL requests, make sure it handles such requests separately from any public IOCTLs or FSCTLs it must also support.

Hiding Devices from Device Manager

6/25/2019 • 2 minutes to read • Edit Online

By default, Device Manager shows the state of every device on a computer. In some situations, you might want to prevent certain devices from appearing in Device Manager. For example, a motherboard might have a CardBus controller with a slot that is not user-accessible. Because the user cannot use the slot, you do not want Device Manager to display any information about the device.

To hide a device in Device Manager, you can mark the device as a *hidden device*. Typically, Device Manager does not display hidden devices. (Note, however, that users can override this setting and display all devices within Device Manager, even hidden ones. For more information about how to override this setting, see Viewing Hidden Devices.)

There are two ways to mark your device as hidden: within the device's driver or by using the ACPI BIOS.

Hiding Devices from Within a Driver

Drivers have two ways to mark a driver as hidden:

- A function driver or function filter driver can ask the operating system to hide a successfully started device by responding to the **IRP_MN_QUERY_PNP_DEVICE_STATE** IRP. When the IRP arrives, the driver must set the PNP_DEVICE_DONT_DISPLAY_UI bit in **IoStatus.Information** to **TRUE** in the driver's dispatch routine.
- On Windows XP and later versions of the Windows operating systems, a bus driver or bus filter driver can hide any device, started or otherwise, by responding to the **IRP_MN_QUERY_CAPABILITIES** IRP. When the IRP arrives, the driver must set the **Parameters.DeviceCapabilities.NoDisplayInUI** member to **TRUE** in the driver's dispatch routine. In some cases, a bus filter driver might have to set this bit in a completion routine. This extra step is required when the underlying bus driver's dispatch routine incorrectly clears all capability fields that other drivers set.

Hiding Devices By Using the ACPI BIOS

You can mark a device as hidden in the ACPI BIOS. The BIOS can expose a _STA method for the device. The _STA method returns a bitmask. Bit 2 (mask 0x4) specifies whether Device Manager should make the device visible by default. This bit should be 1 if the device should be made visible and 0 otherwise.

For example, the following code example shows how a USB controller on the root bus would be hidden.

```
Device(PCI0) // Root PCI bus
_HID *PNP0A03
...
    Device(UCTL)  // USB controller
    _ADR 0xddddffff // dddd = device, ffff = function
    _STA 0xB // Device present, but not shown
```

In Microsoft Windows 2000, you can hide only started, working devices. In Windows XP and later versions of Windows, you can also hide broken devices. Bit 3 (mask 0x8) that is returned by the _STA method indicates whether a device is working properly. This bit is 1 if the device is working properly and is 0 otherwise. For example, the following code example shows how a BIOS would indicate a USB controller is broken and should be hidden:

```
Device(PCI0) // Root PCI bus
_HID *PNP0A03
...
    Device(UCTL) // USB controller
    _ADR 0xddddffff //  dddd = device, ffff = function
    _STA 0x3 // Present, but broken and not shown
```

Note The "decoding" bit (0x2) does not have any relevance for devices that are described through _ADR methods. The previous code examples also work without the decoding bit set. BIOS writers must track the decoding state only for devices that are described through _HID methods.

Filtering Registry Calls

6/25/2019 • 2 minutes to read • Edit Online

A *registry filtering driver* is any kernel-mode driver that filters registry calls, such as the driver component of an antivirus software package. The configuration manager, which implements the registry, allows registry filtering drivers to filter any thread's calls to registry functions. Filtering of registry calls was first supported in Microsoft Windows XP.

On Windows XP, a registry filtering driver can call **CmRegisterCallback** to register a *RegistryCallback* routine and **CmUnRegisterCallback** to unregister the callback routine. The *RegistryCallback* routine receives notifications of each registry operation before the configuration manager processes the operation. A set of **REG_*XXX*_KEY_INFORMATION** data structures contain information about each registry operation. The *RegistryCallback* routine can block a registry operation. The callback routine also receives notifications when the configuration manager has finished creating or opening a registry key.

Windows Server 2003 provides additional completion notifications.

Windows Vista provides the following additional registry filtering capabilities:

- Registry filtering drivers can be layered in a driver stack, and each driver in the stack can filter a registry operation.
- The **CmRegisterCallback** routine is replaced by the **CmRegisterCallbackEx** routine.
- Drivers can completely process a registry operation (or redirect the requested operation to a different operation) and prevent the configuration manager from handling the operation.
- Drivers can assign context information to individual registry operations or key objects.
- Drivers can modify a registry operation's output parameters and return value.
- Additional members have been added to all **REG_*XXX*_KEY_INFORMATION** data structures.
- Drivers receive notifications of additional registry operations.

For a list of the registry operations that a driver can filter on each version of Windows, see **REG_NOTIFY_CLASS**.

To learn more about filtering registry calls, see the following topics:

Registering for Notifications

Handling Notifications

Supporting Layered Registry Filtering Drivers

Specifying Context Information

Obtaining Additional Registry Information

Invalid Key Object Pointers in Registry Notifications

Filtering Registry Operations on Application Hives

Registering for Notifications

6/25/2019 • 2 minutes to read • Edit Online

To filter registry calls, your kernel-mode registry filtering driver must first call **CmRegisterCallback** or **CmRegisterCallbackEx** to register a **RegistryCallback** routine. (For Windows Vista and later operating system versions, drivers should use **CmRegisterCallbackEx** instead of **CmRegisterCallback**.)

After your driver has registered a *RegistryCallback* routine, the configuration manager calls the routine each time that a thread attempts to perform a registry operation. Threads that perform registry operations can be from user-mode applications that call the user-mode registry routines (**RegCreateKeyEx**, **RegOpenKeyEx**, and so on) and from drivers that call the kernel-mode registry routines (**ZwCreateKey**, **ZwOpenKey**, and so on).

For most operations, your driver can receive notification before the configuration manager processes the registry operation (a *pre-notification*) or immediately after the operation completes (but before the configuration manager returns to the caller—a *post-notification*). For a list of the types of notifications that your driver can receive, see **REG_NOTIFY_CLASS**.

After a driver has called **CmRegisterCallback** or **CmRegisterCallbackEx**, the driver will receive notifications until it calls **CmUnRegisterCallback** or is unloaded.

Handling Notifications

6/25/2019 • 2 minutes to read • Edit Online

The *RegistryCallback* routine receives a pointer to a **REG_*XXX*_KEY_INFORMATION** structure that contains information about the registry operation that is occurring.

The *RegistryCallback* routine can monitor, block, or modify a registry operation.

Monitoring Registry Calls

If a registry filtering driver is monitoring registry operations, its *RegistryCallback* routine can update counters or perform other bookkeeping operations and then return STATUS_SUCCESS. Whenever a *RegistryCallback* routine returns STATUS_SUCCESS, the configuration manager continues performing the registry operation.

Monitoring registry calls is supported in Windows XP and later versions of Windows.

Blocking Registry Calls

A registry filtering driver can block registry operations if its *RegistryCallback* routine returns a status value for which NT_SUCCESS(*status*) equals **FALSE** (that is, a non-success NTSTATUS value). When the configuration manager receives a non-success return value, it immediately returns to the calling thread with the driver-specified status value. Therefore, a registry filtering driver can use pre-notifications to prevent registry operations from being processed.

If a *RegistryCallback* routine returns a status value for which NT_SUCCESS(*status*) equals **FALSE** for a pre-notification, the operation's post-notification callback does not occur.

Blocking registry calls is supported in Windows XP and later versions of Windows. For Windows Vista and later, the driver can modify the values that the registry operation returns to the calling thread. These values are contained in the **REG_*XXX*_KEY_INFORMATION** structures for Windows Vista and later.

Modifying Registry Calls

A registry filtering driver can modify a registry operation's output parameters or return value. Additionally, the driver can completely process a registry operation instead of allowing the registry to handle the operation.

When a registry filtering driver's *RegistryCallback* routine receives a post-notification, it can:

- Modify the output parameters that its **REG_*XXX*_KEY_INFORMATION** structure contains and then return STATUS_SUCCESS. The configuration manager returns the modified output parameters to the calling thread.

  Modifying output parameters is supported in Windows Vista and later.

- Modify the registry operation's return value by providing a status value for the **ReturnStatus** member of the **REG_POST_OPERATION_INFORMATION** structure and then returning STATUS_CALLBACK_BYPASS. The configuration manager returns the specified return value to the calling thread.

  Note If the driver changes a status code from success to failure, it might have to deallocate objects that the configuration manager allocated. Alternatively, if the driver changes a status code from failure to success, it might have to provide appropriate output parameters.

Modifying return values is supported in Windows Vista and later.

When a registry filtering driver's *RegistryCallback* routine receives a pre-notification, the routine can handle the registry operation itself and then return STATUS_CALLBACK_BYPASS. When the registry receives

STATUS_CALLBACK_BYPASS from the driver, it just returns STATUS_SUCCESS to the calling thread and does not process the operation. The driver preempts the registry operation and must completely handle it, and the driver must be careful to return valid output values in the **REG_*XXX*_KEY_INFORMATION** structure.

Drivers can preempt registry operations in Windows Vista and later.

If a *RegistryCallback* routine returns STATUS_CALLBACK_BYPASS for a pre-notification, the operation's post-notification callback does not occur.

Note Several registry system calls are not documented because they are rarely used, and, when they are used, it is usually to achieve some unconventional result in the registry. Modifying the operations performed by these calls is difficult and error-prone. Driver developers are discouraged from trying to modify the following registry system calls:

- **NtRestoreKey**
- **NtSaveKey**
- **NtSaveKeyEx**
- **NtLoadKeyEx**
- **NtUnloadKey2**
- **NtUnloadKeyEx**
- **NtReplaceKey**
- **NtRenameKey**
- **NtSetInformationKey**

Supporting Layered Registry Filtering Drivers

6/25/2019 • 2 minutes to read • Edit Online

Windows Vista and later operating system versions support a layered stack of registry filtering drivers. Each driver in the stack can participate in filtering registry operations by registering a *RegistryCallback* routine. Each registry filtering driver is assigned an *altitude*, and drivers can register only one *RegistryCallback* routine for each altitude. When your driver calls **CmRegisterCallbackEx**, the driver specifies its altitude. For more information about altitudes, see Load Order Groups and Altitudes for Minifilter Drivers.

When a thread makes a registry call, the configuration manager calls each *RegistryCallback* routine, in order, from the highest altitude to the lowest, until all drivers have been called or a *RegistryCallback* routine returns a status value for which NT_SUCCESS(*status*) equals **FALSE**. Therefore, if a higher-level driver blocks or modifies a registry operation, the lower-level drivers are not called. (If a driver modifies an operation by calling a different registry function, the configuration manager does not restart at the top of the filter stack.)

Registry filtering drivers that were written before Windows Vista and therefore do not have an altitude assignment are inserted near the top of the Windows Vista filter stack, in the order that they call **CmRegisterCallback**.

Specifying Context Information

6/25/2019 • 2 minutes to read • Edit Online

The configuration manager provides several ways for registry filtering drivers to assign context information to registry operations. A registry filtering driver can:

- Assign context information to the *RegistryCallback* routine.

  When your driver calls **CmRegisterCallback** or **CmRegisterCallbackEx** to register for notification of a registry operation, the driver can specify a driver-defined context value. The configuration manager passes this context value to the driver's *RegistryCallback* routine each time that the configuration manager calls the routine.

  This context information is supported starting with Windows XP.

- Assign context information to a registry operation.

  Drivers can store operation-specific context information in the **CallContext** member of each **REG_*XXX*_KEY_INFORMATION** structure that the driver's *RegistryCallback* routine receives. If your driver receives both a pre-notification and a post-notification for a registry operation, the **REG_POST_OPERATION_INFORMATION** structure contains a pointer to the appropriate pre-notification structure. When a *RegistryCallback* routine receives a **REG_POST_OPERATION_INFORMATION** structure, the **CallContext** member of that structure matches the **CallContext** member of the pre-notification structure.

  The **CallContext** member of these structures is available starting with Windows Vista.

- Assign context information to a registry key object.

  A *RegistryCallback* routine can assign context information to a particular registry key object. If the *RegistryCallback* routine calls **CmSetCallbackObjectContext** to assign context information to a key object, subsequent pre-notifications and post-notifications for all operations on the object will include the context value in the **ObjectContext** member of each **REG_*XXX*_KEY_INFORMATION** structure. If a driver provides multiple *RegistryCallback* routines, the driver can assign different context information for each routine, for a single registry key object.

  If a driver has called **CmSetCallbackObjectContext**, the driver's *RegistryCallback* routine will receive a **RegNtCallbackObjectContextCleanup** notification after the key object's handle has been closed. In response to this notification, the routine should release any resources that it allocated for the object's context. When the *Argument1* parameter to the *RegistryCallback* routine is **RegNtCallbackObjectContextCleanup**, the *Argument2* parameter is a pointer to a **REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION** structure that contains a pointer to the context.

  The **CmSetCallbackObjectContext** routine and **RegNtCallbackObjectContextCleanup** notification are available starting with Windows Vista.

Obtaining Additional Registry Information

6/25/2019 • 2 minutes to read • Edit Online

Registry filtering drivers that run on Windows Vista and later operating system versions can obtain the following additional information about registry operations:

- Object identifiers and names

  The **CmCallbackGetKeyObjectIDEx** routine retrieves the registry key identifier and object name that are associated with a specified registry key object.

- Transaction objects

  The **CmGetBoundTransaction** routine returns a pointer to the transaction object that represents the transaction, if any, that is associated with a registry key object.

- Version information

  The **CmGetCallbackVersion** routine retrieves the major and minor version numbers for the current version of the configuration manager's registry callback feature.

Invalid Key Object Pointers in Registry Notifications

6/25/2019 • 2 minutes to read • Edit Online

To avoid fatal errors and possible memory corruption, a registry filtering driver must not try to access a key object by using an invalid object pointer. This topic lists the circumstances in which the **Object** member of a registry callback notification structure might contain an undefined, non-**NULL** value.

In a registry filtering driver, the second parameter of the *RegistryCallback* routine is a **REG_NOTIFY_CLASS** enumeration value. This value indicates which type of registry callback notification structure the third parameter of the *RegistryCallback* routine points to. The notification structure contains information about the registry operation. The type of this structure varies according to the registry operation that is being performed.

Many of the notification structure types contain an **Object** member that points to a key object. In some cases, the **Object** member can contain a value that is non-**NULL**, but is not a pointer to a valid key object.

Key Object Value is Undefined

If the second parameter in a call to the *RegistryCallback* routine of a registry filtering driver is a **REG_NOTIFY_CLASS** enumeration value of **RegNtPostCreateKeyEx** or **RegNtPostOpenKeyEx**, the third parameter is a pointer to a **REG_POST_OPERATION_INFORMATION** structure. The **Object** member of this structure is valid only if the **Status** member of the structure is set to STATUS_SUCCESS. Any other **Status** value, including a nonzero status code for which the **NT_SUCCESS** macro evaluates to **TRUE**, indicates that the value of the **Object** member is undefined.

Key Object Value is Not in a Valid State

If the second parameter in a registry callback is one of the following **REG_NOTIFY_CLASS** enumeration values, the **Object** member of the registry callback notification structure points to a key object that is being destroyed and whose reference count is zero:

- **RegNtPreKeyHandleClose** (**REG_KEY_HANDLE_CLOSE_INFORMATION** structure)
- **RegNtPostKeyHandleClose** (**REG_POST_OPERATION_INFORMATION** structure)
- **RegNtCallbackObjectContextCleanup** (**REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION** structure)

Because the **Object** member points to a key object that is not in a valid state, the registry filtering driver must not pass the **Object** pointer value as a parameter to a Windows driver support routine (for example, **ObReferenceObjectByPointer**).

However, during a *RegistryCallback* call to handle a **RegNtPreKeyHandleClose** or **RegNtPostKeyHandleClose** notification, a registry filter driver can call a configuration manager routine (for example, **CmGetBoundTransaction**) that takes a registry object as a parameter.

Filtering Registry Operations on Application Hives

6/25/2019 • 2 minutes to read • Edit Online

Initial support for application hives was introduced in Windows Vista. Starting with Windows 8, improved support for application hives is available, and wider use of application hives is expected. Thus, registry filter drivers developed for these versions of Windows, and particularly for Windows 8 and later, must be aware of registry operations on application hives. These drivers should handle such operations efficiently to avoid negatively impacting the user experience.

Application hives are registry hives loaded by user-mode applications to store application-specific state data. An application calls the **RegLoadAppKey** function to load an application hive.

In contrast to other types of registry hives, application hives are loaded under the \REGISTRY\A registry path name instead of under \REGISTRY\MACHINE or \REGISTRY\USER. The \REGISTRY\A path name is special in that there is no way to traverse this path, and an attempt to open a key under \REGISTRY\A will fail with error status STATUS_ACCESS_DENIED. The only way an application can access a key in an application hive is to use the handle to the root key of the application hive. The application gets this handle from the **RegLoadAppKey** call that loads the hive.

An application does not need to explicitly unload an application hive. The operating system automatically unloads the application hive after all of the handles to the hive are closed.

An application hive does not support setting security descriptors on the keys in the hive. Instead, there is one security descriptor for the entire hive. An attempt to set a security descriptor on a key in an application hive will fail with error status STATUS_ACCESS_DENIED. In contrast to other types of registry hives, for which each key is secured with its own security descriptor, the security of an application hive is based on the hive file's security descriptor. Thus, an entity that is successful in loading the hive can modify the entire hive.

A registry filter driver receives calls to its *RegistryCallback* routine for registry operations on application hives. These calls do not distinguish between registry operations on application hives and operations on other types of registry hives. Registry filter drivers that handle create-key and open-key operations (which are indicated by the **RegNtPreOpenKey**, **RegNtPreOpenKeyEx**, **RegNtPreCreateKey**, and **RegNtPreCreateKeyEx** notification values) must correctly handle the following special situation. When an application hive is loaded, the last step in the loading process is the opening of the root key of the hive by the registry manager. The registry manager issues this open-key operation with an absolute path to the key, which means that the path name string in the **CompleteName** member of the **REG_CREATE_KEY_INFORMATION**, **REG_CREATE_KEY_INFORMATION_V1**, **REG_OPEN_KEY_INFORMATION**, or **REG_OPEN_KEY_INFORMATION_V1** structure will start with "\REGISTRY\A\". Only the registry manager can use an absolute path to open an application hive. If a registry filter driver tries to open an application hive in this way (for example, by calling the **ZwOpenKey** routine), the operation will fail with error status STATUS_ACCESS_DENIED.

Object Reference Tracing with Tags

6/25/2019 • 4 minutes to read • Edit Online

Kernel objects are primitive data objects that the Windows kernel implements in system memory to represent the various parts of the computing environment that are managed by the operating system. Kernel objects represent features such as devices, drivers, files, registry keys, events, semaphores, processes, and threads.

Most kernel objects are not permanent. To prevent a nonpermanent kernel object from being deleted while a kernel-mode driver uses the object, the driver can obtain a counted reference to the object. When the driver no longer needs the object, the driver releases its reference to the object.

If a driver does not release all its references to an object, the object's reference count never decrements to zero and the object is never deleted. Therefore, the system resources that are used by the object (for example, system memory) are *leaked*. That is, they cannot be used until the next time that the operating system starts.

Another type of reference error occurs if a driver *under references* an object. In this case, the driver releases more references to an object than the driver actually holds. This error can cause the object to be deleted prematurely, while other clients are still trying to access the object.

Leaks and under-references of kernel objects can be difficult bugs to track down. For example, a process object or a device object might have tens of thousands of references. It can be difficult to identify the source of an object reference bug in these circumstances.

In Windows 7 and later versions of Windows, object references can be tagged to make these bugs easier to find. The following routines associate tags with the acquisition and release of references to kernel objects:

ObDereferenceObjectDeferDeleteWithTag

ObDereferenceObjectWithTag

ObReferenceObjectByHandleWithTag

ObReferenceObjectByPointerWithTag

ObReferenceObjectWithTag

For example, **ObReferenceObjectWithTag** and **ObDereferenceObjectWithTag**, which are available in Windows 7 and later versions of Windows, are enhanced versions of the **ObReferenceObject** and **ObDereferenceObject** routines, which are available in Windows 2000 and later versions of Windows. These enhanced routines enable you to supply a four-byte, custom tag value as an input parameter. The tag value for each call is added to an object reference trace that can be accessed by the Windows debugging tools. **ObReferenceObject** and **ObDereferenceObject** do not enable the caller to specify custom tags, but, in Windows 7 and later versions of Windows, these routines add default tags (with tag value "Dflt") to the trace. Therefore, a call to **ObReferenceObject** or **ObDereferenceObject** has the same effect as a call to **ObReferenceObjectWithTag** or **ObDereferenceObjectWithTag** that specifies a tag value of "Dflt". (In your program, this tag value is represented as 0x746c6644 or 'tlfD'.)

To track down a potential object leak or under-reference, identify a set of associated **ObReferenceObject*Xxx*WithTag** and **ObDereferenceObject*Xxx*WithTag** calls in your driver that increment and decrement the reference count of a particular object. Choose a common tag value (for example, "Lky8") to use for all the calls in this set. After your driver is finished using the object, the number of decrements should match the number of increments exactly. If these numbers do not match, your driver has an object reference bug. The debugger can compare the number of increments and decrements for each tag value and tell you if they do not match. With this capability, you can quickly pinpoint the source of the reference-count mismatch.

To view an object reference trace in the Windows debugging tools, use the !obtrace kernel-mode debugger extension. In Windows 7 and later versions of Windows, the !obtrace extension can display object reference tags, if object reference tracing is enabled. By default, object reference tracing is disabled. Use the Global Flags Editor (Gflags) to enable object reference tracing. For more information about Gflags, see Configuring Object Reference Tracing.

After object reference tracing is enabled, the output that is produced by the !obtrace extension includes a "Tag" column, as the following example shows:

```
0: kd> !obtrace 0x8a226130
Object: 8a226130
 Image: leakyapp.exe
Sequence   (+/-)   Tag    Stack
--------   -----   ----   ----------------------------------------------
      36    +1     Dflt      nt!ObCreateObject+1c4
                             nt!NtCreateEvent+93
                             nt!KiFastCallEntry+12a

      37    +1     Dflt      nt!ObpCreateHandle+1c1
                             nt!ObInsertObjectEx+d8
                             nt!ObInsertObject+1e
                             nt!NtCreateEvent+ba
                             nt!KiFastCallEntry+12a

      38    -1     Dflt      nt!ObfDereferenceObjectWithTag+22
                             nt!ObInsertObject+1e
                             nt!NtCreateEvent+ba
                             nt!KiFastCallEntry+12a

      39    +1     Lky8      nt!ObReferenceObjectByHandleWithTag+254
                             leakydrv!LeakyCtlDeviceControl+6c
                             nt!IofCallDriver+63
                             nt!IopSynchronousServiceTail+1f8
                             nt!IopXxxControlFile+6aa
                             nt!NtDeviceIoControlFile+2a
                             nt!KiFastCallEntry+12a

      3a    -1     Dflt      nt!ObfDereferenceObjectWithTag+22
                             nt!ObpCloseHandle+7f
                             nt!NtClose+4e
                             nt!KiFastCallEntry+12a

--------   -----   ----   ----------------------------------------------
References: 3, Dereferences 2
Tag: Lky8 References: 1 Dereferences: 0 Over reference by: 1
```

The last line in this example indicates that the reference and dereference counts that are associated with the "Lky8" tag do not match and that the result of this mismatch is an over-reference by one (that is, a leak).

If the result were instead an under-reference, the last line of the !obtrace output might be as follows:

```
Tag: Lky8 References: 1 Dereferences: 2 Under reference by: 1
```

By default, the operating system conserves memory by deleting the object reference trace for an object after the object is freed. To track down an under-reference requires that the trace remain stored in memory even after the object is freed. For this purpose, the Gflags tool provides a "Permanent" option, which preserves the trace in memory while the computer shuts down and starts again.

Object reference tracing, without tags, was introduced in Windows XP. Because the trace did not include tags, developers had to use less convenient techniques to identify object reference bugs. The debugger could track the references of groups of objects, which the developer selected by object type. The only way that the developer could

identify the various sources of object references and dereferences was to compare their call stacks. Although the previous !obtrace example contains only five stacks, certain types of object, such as a process (**EPROCESS**) object, might be referenced and dereferenced many thousands of times. With thousands of stacks to inspect, it might be difficult to identify the source of an object leak or under-reference without using tags.

Porting Your Driver to 64-Bit Windows

12/5/2018 • 2 minutes to read • Edit Online

The 64-bit version of Windows is designed to make it possible for developers to use a single source-code base for their 32-bit and 64-bit Windows applications. To a large extent, this is also true for 32-bit and 64-bit Windows drivers.

For user-mode applications, 64-bit Windows includes a Windows on Windows (WOW64) *thunking layer* that enables 32-bit applications to execute (with some performance degradation) on 64-bit versions of Windows. It does this by intercepting 32-bit function calls and converting pointer-precision parameter types to fixed-precision types as appropriate before making the transition to the 64-bit kernel. This conversion process is called *thunking*.

Note This thunking is only done for 32-bit *applications*; 32-bit *drivers* are not supported on 64-bit versions of Windows.

What's Changed

12/5/2018 • 2 minutes to read • Edit Online

On 32-bit Windows, the integer, long, and pointer data types are all the same size—32 bits. This convenient uniformity in data type sizes has been a boon to clever C programmers, many of whom have come to take it for granted.

On 64-bit Windows, however, this assumption of uniformity is no longer valid. Pointers are now 64 bits in length, but integer and long data types remain the same size as before—32 bits. This is because, while 64-bit pointers are needed to accommodate systems with as much as 16 TB of virtual memory, most data still fits comfortably into 32-bit integers. For most applications, changing the default integer size to 64 bits would only be a waste of space.

On 32-bit Windows platforms, the operating system automatically fixes kernel-mode memory alignment faults and makes them invisible to the application. It does this for the calling process and any descendant processes. This feature, which often dramatically reduces performance, has not been implemented in 64-bit Windows. Thus, if your 32-bit driver contains misalignment bugs, you will need to fix them when porting to 64-bit Windows.

The New Data Types

12/5/2018 • 3 minutes to read • Edit Online

There are three classes of new data types: fixed-precision integer types, pointer-precision integer types, and specific-precision pointer types. These types were added to the Windows environment (specifically, to Basetsd.h) to allow developers to prepare for 64-bit Windows well before its introduction. These new types were derived from the basic C-language integer and long types, so they work in existing code. Therefore, use these data types in your code now, test your code on 32-bit Windows, and use the 64-bit compiler to find and fix portability problems in advance, so your driver can be ready when 64-bit Windows is available for testing.

In addition, adopting these new data types will make your code more robust. To use these data types, you must scan your code for potentially unsafe pointer usage, polymorphism, and data definitions. To be safe, use the new types. For example, when a variable is of type **ULONG_PTR**, it is clear that it will be used for casting pointers for arithmetic operations or polymorphism. It is not possible to indicate such usage directly by using the native Win32 data types. You can do this by using derived type naming or Hungarian notation, but both techniques are prone to errors.

Fixed-Precision Integer Types

Fixed-precision data types are the same length for 32-bit and 64-bit programming. To help you remember this, their precision is part of the name of the data type. The following are the fixed-precision data types.

TYPE	DEFINITION
DWORD32	32-bit unsigned integer
DWORD64	64-bit unsigned integer
INT32	32-bit signed integer
INT64	64-bit signed integer
LONG32	32-bit signed integer
LONG64	64-bit signed integer
UINT32	Unsigned **INT32**
UINT64	Unsigned **INT64**
ULONG32	Unsigned **LONG32**
ULONG64	Unsigned **LONG64**

Pointer-Precision Integer Types

As the pointer precision changes (that is, as it becomes 32 bits when compiled for 32-bit platforms, 64 bits when compiled for 64-bit platforms), these data types reflect the precision accordingly. Therefore, it is safe to cast a pointer to one of these types when performing pointer arithmetic; if the pointer precision is 64 bits, the type is 64 bits. The count types also reflect the maximum size to which a pointer can refer. The following are the pointer-precision and count types.

TYPE	DEFINITION
DWORD_PTR	Unsigned long type for pointer precision.
HALF_PTR	Signed integral type for half-pointer precision (16 bits on 32-bit systems, 32 bits on 64-bit systems).
INT_PTR	Signed integral type for pointer precision.
LONG_PTR	Signed long type for pointer precision.
SIZE_T	The maximum number of bytes to which a pointer can refer. Use this type for a count that must span the full range of a pointer.
SSIZE_T	Signed **SIZE_T**.
UHALF_PTR	Unsigned **HALF_PTR**.
UINT_PTR	Unsigned **INT_PTR**.
ULONG_PTR	Unsigned **LONG_PTR**.

Fixed-Precision Pointer Types

There are also new pointer types that explicitly size the pointer. Be cautious when using these pointer types in 64-bit code: If you declare the pointer using a 32-bit type, the system creates the pointer by truncating a 64-bit pointer.

TYPE	DEFINITION
POINTER_32	A 32-bit pointer. On a 32-bit system, this is a native pointer. On a 64-bit system, this is a truncated 64-bit pointer.
POINTER_64	A 64-bit pointer. On a 64-bit system, this is a native pointer. On a 32-bit system, this is a sign-extended 32-bit pointer. Note that it is not safe to assume the state of the high pointer bit.

Helper Functions

The following inline functions (defined in Basetsd.h) can help you safely convert values from one type to another:

```
unsigned long HandleToUlong( const void *h )
long HandleToLong( const void *h )
void * LongToHandle( const long h )
unsigned long PtrToUlong( const void *p )
unsigned int PtrToUint( const void *p )
unsigned short PtrToUshort( const void *p )
long PtrToLong( const void *p )
int PtrToInt( const void *p )
short PtrToShort( const void *p )
void * IntToPtr( const int i )
void * UIntToPtr( const unsigned int ui )
void * LongToPtr( const long l )
void * ULongToPtr( const unsigned long ul )
```

Warning **IntToPtr** sign-extends the **int** value, **UIntToPtr** zero-extends the unsigned **int** value, **LongToPtr** sign-extends the **long** value, and **ULongToPtr** zero-extends the **unsigned long** value.

64-Bit Compiler

3/5/2019 • 2 minutes to read • Edit Online

After you convert your 32-bit driver source code to use the new data types, you can use the 64-bit compiler to identify any type-related problems that you initially missed. The first time you compile this code for 64-bit Windows, the compiler might generate many pointer-truncation or type-mismatch warnings. Use these warnings as a guide to make your code more robust. It is good practice to eliminate all warnings, especially pointer-truncation warnings.

The following is an example of such a warning:

```
warning C4311: 'type cast' : pointer truncation from 'unsigned char *' to 'unsigned long '
```

For example, the following code can generate the C4311 warning:

```
buffer = (PUCHAR)srbControl;
(ULONG)buffer += srbControl->HeaderLength;
```

To correct the code, make the following changes:

```
buffer = (PUCHAR)srbControl;
(ULONG_PTR)buffer += srbControl->HeaderLength;
```

Predefined macros

The compiler defines the following macros to identify the platform.

MACRO	MEANING
_WIN64	A 64-bit platform.
_WIN32	A 32-bit platform. This value is also defined by the 64-bit compiler for backward compatibility.
_WIN16	A 16-bit platform.

The following macros are specific to the architecture.

MACRO	MEANING
_M_IA64	A 64-bit Intel platform.
_M_IX86	A 32-bit Intel platform.

Do not use these macros except with architecture-specific code. Instead, use _WIN64, _WIN32, and _WIN16 whenever possible.

64-Bit compiler switches and warnings

There is a warning option to assist porting to 64-bit Windows. The -Wp64-W3 switch enables the following warnings:

- **C4305**: Truncation warning. For example, "return": truncation from "unsigned int64" to "long."
- **C4311**: Truncation warning. For example, "type cast": pointer truncation from "int*_ptr64" to "int."
- **C4312**: Conversion to bigger-size warning. For example, "type cast": conversion from "int" to "int*_ptr64" of greater size.
- **C4318**: Passing zero length. For example, passing constant zero as the length to the **memset** function.
- **C4319**: Not operator. For example, "~": zero extending "unsigned long" to "unsigned _int64" of greater size.
- **C4313**: Calling the **printf** family of functions with conflicting conversion type specifiers and arguments. For example, "printf": "%p" in format string conflicts with argument 2 of type "_int64." Another example is the call printf("%x", pointer_value); this causes a truncation of the upper 32 bits. The correct call is printf("%p", pointer_value).
- **C4244**: Same as the existing warning C4242. For example, "return": conversion from "_int64" to "unsigned int," possible loss of data.

Performing DMA in 64-Bit Windows

6/25/2019 • 2 minutes to read • Edit Online

Adding 64-bit addressing support to your driver can significantly improve overall system performance. This is particularly important for device drivers that perform direct memory access (DMA). In 64-bit Microsoft Windows, device drivers that perform DMA but do not support 64-bit addressing are double-buffered, which results in lower relative performance.

Although double-buffering usually has a relatively small impact (single percentage points) on 8 GB systems, this is enough to impact I/O-intensive tasks, such as database activity. As the amount of physical memory increases, this negative performance impact increases as well.

To support 64-bit DMA, drivers should observe the following guidelines:

1. Use **PHYSICAL_ADDRESS** structures for physical address calculations.
2. Treat the entire 64-bit address as a valid physical address. For example, drivers should not call **MmGetPhysicalAddress** on a locked buffer, discard the high 32 bits, and pass the truncated address to a 32-bit component adapter. This results in corrupted memory, lost I/O, and system failure.
3. Use the high-performance scatter/gather routines (**GetScatterGatherList** and **PutScatterGatherList**) that were added in Windows 2000.
4. Check the value of the **Mm64BitPhysicalAddress** global system variable. If it is **TRUE**, the system supports 64-bit physical addressing.
5. Set the **Dma64BitAddresses** member of the **DEVICE_DESCRIPTION** structure to **TRUE** to indicate that your driver supports 64-bit DMA addresses.

The DMA routines in 32-bit Windows are 64-bit-ready. If your device driver uses these routines correctly, your DMA code should work without modification on 64-bit Windows.

Using extended processor features in Windows drivers

6/25/2019 • 2 minutes to read • Edit Online

Last updated

- July 2016

Windows drivers for x86 and x64 systems that use extended processor features must wrap floating point calculations between calls to **KeSaveExtendedProcessorState** and **KeRestoreExtendedProcessorState** in order to avoid errors in concurrent applications that might be using the registers.

Legacy MMX/x87 Registers

These registers correspond to the XSTATE_MASK_LEGACY_FLOATING_POINT mask and are unavailable to drivers for x64 systems. For more information on these registers see Using Floating Point in a WDM Driver.

SSE Registers

These registers correspond to the XSTATE_MASK_LEGACY_SSE flag and are used by the x64 compiler for floating point operations. Drivers for x86 systems that use these registers must save them before use by passing the XSTATE_MASK_LEGACY or XSTATE_MASK_LEGACY_SSE flag in the **KeSaveExtendedProcessorState** call and when finished, restore them with **KeRestoreExtendedProcessorState**. This is unnecessary on x64 systems, but not harmful. For more information about these registers see Using Floating Point in a WDM Driver.

AVX Registers

These registers correspond to the XSTATE_MASK_GSSE or XSTATE_MASK_AVX masks. New x86 processors, such as the Intel Sandy Bridge (formerly Gesher) processor, support the AVX instructions and register set (YMM0-YMM15). In Windows 7 with Service Pack 1 (SP1), Windows Server 2008 R2, and newer versions of Windows, both x86 and x64 versions of the operating system preserve the AVX registers across thread (and process) switches. To use the AVX registers in kernel mode, drivers (x86 and x64) must explicitly save and restore the AVX registers. AVX registers cannot be used in an interrupt service routine, and arithmetic exceptions are turned off by default.

```
include ksamd64.inc

        subttl "Set YMM State."
;++
;
; Routine Description:
;
;   This routine loads the first four YMM registers with the state supplied.
;
; Arguments;
;
;   rcx - Supplies a pointer to the values we want to load.
;
; Return Value:
;
;   None
;
;--

LEAF_ENTRY SetYmmValues, _TEXT$00

        vmovdqa    ymm0,  ymmword ptr[rcx + 0]
        vmovdqa    ymm1,  ymmword ptr[rcx + 32]
        vmovdqa    ymm2,  ymmword ptr[rcx + 64]
        vmovdqa    ymm3,  ymmword ptr[rcx + 96]

        ret

LEAF_END SetYmmValues, _TEXT$00

        end
```

```
typedef DECLSPEC_ALIGN(32) struct _YMM_REGISTERS {
    ULONG64 Ymm4Registers[16];
} YMM_REGISTERS, *PYMM_REGISTERS;

VOID
FASTCALL
SetYmmValues(
    __in PYMM_REGISTERS YmmRegisterValues
    );

NTSTATUS
DriverEntry (
    __in PDRIVER_OBJECT DriverObject,
    __in PUNICODE_STRING RegistryPath
    )
{

    NTSTATUS Status;
    XSTATE_SAVE SaveState;
    ULONG64 EnabledFeatures;

    //
    // Load the first 4 YMM registers as 4 vectors of 4 64-bit integers.
    //

    YMM_REGISTERS RegisterValues = { 0, 1, 2, 3,        // YMM0
                                     4, 5, 6, 7,        // YMM1
                                     8, 9, 10, 11,      // YMM2
                                     12, 13, 14, 15 };  // YMM3

    //
    // Check to see if AVX is available. Bail if it is not.
    //

    EnabledFeatures = RtlGetEnabledExtendedFeatures(-1);
    if ((EnabledFeatures & XSTATE_MASK_GSSE) == 0) {
        Status = STATUS_FAILED_DRIVER_ENTRY;
        goto exit;
    }

    Status = KeSaveExtendedProcessorState(XSTATE_MASK_GSSE, &SaveState);

    if (!NT_SUCCESS(Status)) {
        goto exit;
    }

    __try {
        SetYmmValues(&RegisterValues);
    }
    __finally {
        KeRestoreExtendedProcessorState(&SaveState);
    }

exit:
    return Status;
}
```

Related topics

KeSaveExtendedProcessorState

KeRestoreExtendedProcessorState

Using Floating Point in a WDM Driver

Porting Issues Checklist

6/25/2019 • 7 minutes to read • Edit Online

General

- Use the new 64-bit-safe Windows data types.

  The new 64-bit-safe data types, described earlier in this document, are defined in Basetsd.h. This header file is included in Ntdef.h, which is included in Ntddk.h, Wdm.h, and Ntifs.h.

- Use the platform compiler macros carefully.

  The following assumption is no longer valid:

```
#ifdef _WIN32  // 32-bit Windows code
...
#else          // 16-bit Windows code
...
#endif
```

  However, the 64-bit compiler defines _WIN32 for backward compatibility.

  Also, the following assumption is no longer valid:

```
#ifdef _WIN16  // 16-bit Windows code
...
#else          // 32-bit Windows code
...
#endif
```

  In this case, the else clause can represent _WIN32 or _WIN64.

- Use the proper format specifiers with **printf** and **wsprintf**.

  Use **%p** to print pointers in hexadecimal. This is the best choice for printing pointers.

  Note A future version of Visual C++ will support **%I** to print polymorphic data. It will treat values as 64 bits in 64-bit Windows and 32 bits in 32-bit Windows. Visual C++ will also support **%I64** to print values that are 64 bits.

- Know your address space.

  Do not blindly assume, for example, that if an address is a kernel address, its high-order bit must be set. To obtain the lowest system address, use the **MM_LOWEST_SYSTEM_ADDRESS** macro.

Pointer Arithmetic

- Be careful when performing unsigned and signed operations.

  Consider the following:

```
ULONG x;
LONG y;
LONG *pVar1;
LONG *pVar2;

pVar2 = pVar1 + y * (x - 1);
```

The problem arises because *x* is unsigned, which makes the entire expression unsigned. This works fine unless *y* is negative. In this case, *y* is converted to an unsigned value, the expression is evaluated using 32-bit precision, scaled, and added to *pVar1*. On 64-bit Windows, this 32-bit unsigned negative number becomes a large 64-bit positive number, which gives the wrong result. To fix this problem, declare *x* as a signed value or explicitly typecast it to **LONG** in the expression.

- Be careful when using hexadecimal constants and unsigned values.

  The following assertion is not true on 64-bit systems:

  ```
  ~((UINT64)(PAGE_SIZE-1)) == (UINT64)~(PAGE_SIZE-1)
  PAGE_SIZE = 0x1000UL  // Unsigned long - 32 bits
  PAGE_SIZE - 1 = 0x00000fff
  ```

  LHS expression:

  ```
  // Unsigned expansion(UINT64)(PAGE_SIZE -1 ) = 0x0000000000000fff
  ~((UINT64)(PAGE_SIZE -1 )) = 0xfffffffffffff000
  ```

  RHS expression:

  ```
  ~(PAGE_SIZE-1) = 0xfffff000
  (UINT64)(~(PAGE_SIZE - 1)) = 0x00000000fffff000
  ```

  Hence:

  ```
  ~((UINT64)(PAGE_SIZE-1)) != (UINT64)(~(PAGE_SIZE-1))
  ```

- Be careful with NOT operations.

  Consider the following:

  ```
  UINT_PTR a; ULONG b;
  a = a & ~(b - 1);
  ```

  The problem is that ~(b−1) produces 0x0000 0000 *xxxx xxxx* and not 0xFFFF FFFF *xxxx xxxx*. The compiler will not detect this. To fix this, change the code as follows:

  ```
  a = a & ~((UINT_PTR)b - 1);
  ```

- Be careful when computing buffer sizes.

  Consider the following:

```
len = ptr2 - ptr1
/* len could be greater than 2**32 */
```

Cast pointers to **PCHAR** for pointer arithmetic.

Note If *len* is declared **INT** or **ULONG**, this will generate a compiler warning. Buffer sizes, even when computed correctly, may still exceed the capacity of **ULONG**.

- Avoid using computed or hard-coded pointer offsets.

  When working with structures, use the **FIELD_OFFSET** macro wherever possible to determine the offset of structure members.

- Avoid using hard-coded pointer or handle values.

  Do not pass hard-coded pointers or handles such as (HANDLE)0xFFFFFFFF to routines such as **ZwCreateSection**. Instead, use constants, such as INVALID_HANDLE_VALUE, that can be defined to have the appropriate value for each platform.

- Be aware that in 64-bit Windows, 0xFFFFFFFF is not the same as -1.

  For example:

```
DWORD index = 0;
CHAR *p;

// if (p[index-1] == '0') causes access violation on 64-bit Windows!
```

  On 32-bit machines:

```
p[index-1] == p[0xffffffff] == p[-1]
```

  On 64-bit machines:

```
p[index-1] == p[0x00000000ffffffff] != p[-1]
```

  This problem can be avoided by changing the type of *index* from **DWORD** to **DWORD_PTR**.

Polymorphism

- Be careful with polymorphic interfaces.

  Do not create functions that accept parameters of type **DWORD** (or other fixed-precision types) for polymorphic data. If the data can be a pointer or an integral value, the parameter type should be **UINT_PTR** or **PVOID**, not **DWORD**.

  For example, do not create a function that accepts an array of exception parameters typed as **DWORD** values. The array should be an array of **DWORD_PTR** values. Therefore, the array elements can hold addresses or 32-bit integral values. The general rule is that if the original type is **DWORD** and it needs to be pointer width, convert it to a **DWORD_PTR** value. That is why there are corresponding pointer-precision types for the native Win32 types. If you have code that uses **DWORD**, **ULONG**, or other 32-bit types in a polymorphic way (that is, you really want the parameter or structure member to hold an address), use **UINT_PTR** in place of the current type.

- Be careful when calling functions that have pointer OUT parameters.

  Do not do this:

```
void GetBufferAddress(OUT PULONG *ptr);
{
  *ptr=0x1000100010001000;
}
void foo()
{
  ULONG bufAddress;
  //
  // This call causes memory corruption.
  //
  GetBufferAddress((PULONG *)&bufAddress);
}
```

Typecasting *bufAddress* to (**PULONG** *) prevents a compiler error. However, *GetBufferAddress* will write a 64-bit value into the memory location at *&bufAddress*. Because *bufAddress* is only a 32-bit value, the 32 bits immediately following *bufAddress* will get overwritten. This is a very subtle, hard-to-find bug.

- Do not cast pointers to **INT**, **LONG**, **ULONG**, or **DWORD**.

  If you must cast a pointer to test some bits, set or clear bits, or otherwise manipulate its contents, use the **UINT_PTR** or **INT_PTR** type. These types are integral types that scale to the size of a pointer for both 32-bit and 64-bit Windows (for example, **ULONG** for 32-bit Windows and **_int64** for 64-bit Windows). For example, assume you are porting the following code:

  ```
  ImageBase = (PVOID)((ULONG)ImageBase | 1);
  ```

  As a part of the porting process, you would change the code as follows:

  ```
  ImageBase = (PVOID)((ULONG_PTR)ImageBase | 1);
  ```

  Use **UINT_PTR** and **INT_PTR** where appropriate (and if you are uncertain whether they are required, there is no harm in using them just in case). Do not cast your pointers to the types **ULONG**, **LONG**, **INT**, **UINT**, or **DWORD**.

  Note HANDLE is defined as a **void **, *so typecasting a *HANDLE* value to a* **ULONG** value to test, set, or clear the low two bits is a programming error.

- Use **PtrToLong** and **PtrToUlong** to truncate pointers.

  If you must truncate a pointer to a 32-bit value, use the **PtrToLong** or **PtrToUlong** function (defined in *Basetsd.h*). This function disables the pointer truncation warning for the duration of the call.

  Use these functions carefully. After you truncate a pointer variable using one of these functions, never cast the resulting **LONG** or **ULONG** back to a pointer. These functions truncate the upper 32 bits of an address, which are usually needed to access the memory originally referenced by pointer. Using these functions without careful consideration will result in fragile code.

Data Structures and Structure Alignment

- Carefully examine all uses of data structure pointers.

  The following are common trouble areas:

  - Data structures that are stored on disk or exchanged with 32-bit processes.
  - Explicit and implicit unions with pointers.
  - Security descriptors.

- Use the **FIELD_OFFSET** macro.

For example:

```
struct xx {
   DWORD NumberOfPointers;
   PVOID Pointers[1];
};
```

The following allocation is incorrect in 64-bit Windows because the compiler will pad the structure with an additional 4 bytes to make the 8-byte alignment requirement:

```
malloc(sizeof(DWORD)+100*sizeof(PVOID));
```

Here is how to do it correctly:

```
malloc(FIELD_OFFSET(struct xx, Pointers) +100*sizeof(PVOID));
```

- Use the **TYPE_ALIGNMENT** macro.

  The **TYPE_ALIGNMENT** macro returns the alignment requirement for a given data type on the current platform. For example:

  ```
  TYPE_ALIGNMENT(KFLOATING_SAVE) == 4 on x86, 8 on Itanium
  TYPE_ALIGNMENT(UCHAR) == 1 everywhere
  ```

  As an example, code such as this:

  ```
  ProbeForRead(UserBuffer, UserBufferLength, sizeof(ULONG));
  ```

  becomes more portable when changed to:

  ```
  ProbeForRead(UserBuffer, UserBufferLength, TYPE_ALIGNMENT(ULONG));
  ```

- Watch for data type changes in public kernel structures.

  For example, the **Information** field in the IO_STATUS_BLOCK structure is now of type **ULONG_PTR**.

- Be cautious when using structure packing directives.

  On 64-bit Windows, if a data structure is misaligned, routines that manipulate the structure, such as **RtlCopyMemory** and **memcpy**, will not fault. Instead, they will raise an exception. For example:

  ```
  #pragma pack (1)  /* also set by /Zp switch */
  struct Buffer {
      ULONG size;
      void *ptr;
  };

  void SetPointer(void *p) {
      struct Buffer s;
      s.ptr = p;  /* will cause alignment fault */
      ...
  }
  ```

You could use the **UNALIGNED** macro to fix this:

```
void SetPointer(void *p) {
    struct Buffer s;
    *(UNALIGNED void *)&s.ptr = p;
}
```

Unfortunately, using the **UNALIGNED** macro is very expensive on Itanium-based processors. A better solution is to put 64-bit values and pointers at the beginning of the structure.

Note If possible, avoid using different packing levels in the same header file.

Additional Information

- Supporting 32-Bit I/O in Your 64-Bit Driver
- Getting Ready for 64-bit Windows (user-mode application porting guide)

Supporting 32-Bit I/O in Your 64-Bit Driver

6/25/2019 • 2 minutes to read • Edit Online

Windows on Windows (WOW64) enables Microsoft Win32 user-mode applications to run on 64-bit Windows. It does this by intercepting Win32 function calls and converting parameters from pointer-precision types to fixed-precision types as appropriate before making the transition to the 64-bit kernel. This conversion, which is called *thunking*, is done automatically for all Win32 functions, with one important exception: the data buffers passed to **DeviceIoControl**. The contents of these buffers, which are pointed to by the *InputBuffer* and *OutputBuffer* parameters, are not thunked, because their structure is driver-specific.

Note Although the buffer *contents* are not thunked, the buffer *pointers* are converted into 64-bit pointers.

User-mode applications call **DeviceIoControl** to send an I/O request directly to a specified kernel-mode driver. This request contains an I/O control code (IOCTL) or file system control code (FSCTL) and pointers to input and output data buffers. The format of these data buffers is specific to the IOCTL or FSCTL, which in turn is defined by the kernel-mode driver. Because the buffer format is arbitrary, and because it is known to the driver and not WOW64, the task of thunking the data is left to the driver.

Your 64-bit driver must support 32-bit I/O if all of the following are true:

- The driver exposes an IOCTL (or FSCTL) to user-mode applications.
- At least one of the I/O buffers used by the IOCTL contains pointer-precision data types.
- Your IOCTL code cannot easily be rewritten to eliminate the use of pointer-precision buffer data types.

Why Thunking Is Necessary

12/5/2018 • 2 minutes to read • Edit Online

Kernel-mode drivers must validate the size of any I/O buffer passed in from a user-mode application. If a 32-bit application passes a buffer containing pointer-precision data types to a 64-bit driver, and no thunking takes place, the driver will expect the buffer to be larger than it actually is. This is because pointer precision is 32 bits on 32-bit Microsoft Windows and 64 bits on 64-bit Windows. For example, consider the following structure definition:

```
typedef struct _DRIVER_DATA
{
    HANDLE           Event;
    UNICODE_STRING   ObjectName;
} DRIVER_DATA;
```

On 32-bit Windows, the size of the DRIVER_DATA structure is 12 bytes.

HANDLE **Event** UNICODE_STRING **ObjectName** USHORT Length USHORT Maximum Length PWSTR Buffer 32 bits (4 bytes) 16 bits (2 bytes) 16 bits (2 bytes) 32 bits (4 bytes)

On 64-bit Windows, the size of the DRIVER_DATA structure is 24 bytes. (The 4 bytes of structure padding are required so that the **Buffer** member can be aligned on an 8-byte boundary.)

HANDLE **Event** UNICODE_STRING **ObjectName** USHORT Length USHORT Maximum Length Empty (Structure Padding) PWSTR Buffer 64 bits (8 bytes) 16 bits (2 bytes) 16 bits (2 bytes) 32 bits (4 bytes) 64 bits (8 bytes)

If a 64-bit driver receives 12 bytes of DRIVER_DATA when it expected 24 bytes, the size validation will fail. To prevent this, the driver must detect whether a DRIVER_DATA structure was sent by a 32-bit application, and if so, thunk it appropriately before performing the validation.

For example, a thunked version of the above DRIVER_DATA structure could be defined as follows:

```
typedef struct _DRIVER_DATA32
{
    VOID *POINTER_32   Event;
    UNICODE_STRING32   ObjectName;
} DRIVER_DATA32;
```

Because it contains only fixed-precision data types, this new structure is the same size on 32-bit Windows and 64-bit Windows.

POINTER_32 **Event** UNICODE_STRING32 **ObjectName** USHORT Length USHORT Maximum Length ULONG Buffer 32 bits (4 bytes) 16 bits (2 bytes) 16 bits (2 bytes) 32 bits (4 bytes)

Which Data Types Need Thunking

12/5/2018 • 2 minutes to read • Edit Online

The following table lists common data types that require thunking, along with their thunked equivalents.

POINTER-PRECISION DATA TYPE (BEFORE THUNKING)	EQUIVALENT 32-BIT FIXED-PRECISION DATA TYPE (AFTER THUNKING)
HANDLE	VOID * POINTER_32
INT_PTR	INT32
LONG_PTR	LONG32
LPARAM	LONG32
PCHAR	Char * POINTER_32
PDWORD	DWORD * POINTER_32
PHANDLE	VOID ** POINTER_32
PULONG	ULONG * POINTER_32
PVOID	VOID * POINTER_32
PWORD	WORD * POINTER_32
SIZE_T	INT32
ULONG_PTR	ULONG32
UNICODE_STRING	UNICODE_STRING32

How Drivers Identify 32-Bit Callers

6/25/2019 • 2 minutes to read • Edit Online

There are two ways for drivers to determine whether the originator of an IOCTL or FSCTL request is a 32-bit or 64-bit application. The first is for the application to identify itself. The second is for the driver to determine on its own whether the application is 32-bit or 64-bit.

The first technique involves defining a "64Bit" field in the IOCTL or FSCTL control code. This field contains a single bit, which is set only for 64-bit callers. Thus 64-bit callers identify themselves by using a separate set of 64-bit control codes in which this bit is set. 32-bit callers use a similar set of control codes in which this bit is not set.

The second technique permits 32- and 64-bit applications to continue using the same IOCTL or FSCTL codes. Instead, the driver determines whether the user-mode process is 32- or 64-bit by calling **IoIs32bitProcess**.

The first technique is more efficient, because the driver checks a bit flag instead of calling a kernel-mode routine. However, the second technique requires no changes to user-mode code. Which technique you should use depends on the requirements of your driver and the applications that send I/O requests to it.

Technique 1: Defining a "64Bit" Field

12/5/2018 • 2 minutes to read • Edit Online

The "64Bit" field is defined in the IOCTL or FSCTL control code. This field contains a bit flag that is always set for 64-bit callers, but is always clear for 32-bit. Which bit in the control code is chosen as the "64Bit" field is driver-specific, but it must be a bit that is never set for 32-bit callers. A good choice for most drivers is the most significant bit (MSB) in the Function field.

For example, the IOCTL (FSCTL) control codes used in 32-bit drivers contain four bitfields:

DEVICE TYPE	ACCESS	FUNCTION	METHOD
16 bits	2 bits	12 bits	2 bits

As long as none of the existing driver-defined control codes set the MSB in the Function field, these control codes can continue to be used by 32-bit user-mode applications.

To accommodate 64-bit callers, the driver defines a Function field that is shorter by one bit. This bit is redefined as a "64Bit" field:

DEVICE TYPE	ACCESS	64BIT	FUNCTION	METHOD
16 bits	2 bits	1 bit	11 bits	2 bits

The following code example shows how to define a "64Bit" field in a driver header file:

```
#define REGISTER_FUNCTION 0     // Define the IOCTL function code

#ifdef  _WIN64
#define CLIENT_64BIT   0x800
#define REGISTER_FUNCTION 0
#define IOCTL_REGISTER   CTL_CODE(FILE_DEVICE_UNKNOWN, \
  CLIENT_64BIT|REGISTER_FUNCTION, METHOD_BUFFERED, FILE_ANY_ACCESS)
#else
#define IOCTL_REGISTER   CTL_CODE(FILE_DEVICE_UNKNOWN, \
  REGISTER_FUNCTION, METHOD_BUFFERED, FILE_ANY_ACCESS)
#endif

typedef struct _IOCTL_PARAMETERS {
    PVOID   Addr;
    SIZE_T  Length;
    HANDLE  Handle;
} IOCTL_PARAMETERS, *PIOCTL_PARAMETERS;
```

Technique 2: Using IoIs32bitProcess

6/25/2019 • 2 minutes to read • Edit Online

In cases where it is not practical to define separate IOCTL or FSCTL control codes for I/O requests from 32-bit and 64-bit applications, it is left to the driver to determine which type of application sent the I/O request. The 64-bit version of Microsoft Windows introduces a new kernel-mode routine, **IoIs32bitProcess**, that detects whether the current I/O request originated in a 32-bit user-mode process. Its prototype is:

```
BOOLEAN
  IoIs32bitProcess(
    _In_opt_ PIRP Irp  // NULL for fast I/O call, IRP otherwise
    );
```

IoIs32bitProcess returns **TRUE** if the originator of the current I/O request is a 32-bit user-mode application.

The following code sample shows how to use **IoIs32bitProcess**:

```
typedef UINT32 POINTER_32 PVOID32;

typedef struct _IOCTL_PARAMETERS
{
    PVOID   Addr;
    SIZE_T  Length;
    HANDLE  Handle;
} IOCTL_PARAMETERS, *PIOCTL_PARAMETERS;

typedef struct _IOCTL_PARAMETERS_32
{
    PVOID32  Addr;
    INT32    Length;
    PVOID32  Handle;
} IOCTL_PARAMETERS_32, *PIOCTL_PARAMETERS_32;

...

IOCTL_PARAMETERS LocalParam;

if (IoIs32bitProcess(Irp))
{
    /* 32-bit process IOCTL */
    PIOCTL_PARAMETERS_32 params32;

    params32 = (PIOCTL_PARAMETERS_32)(Irp->AssociatedIrp.SystemBuffer);
    if (irpSp->Parameters.DeviceIoControl.InputBufferLength < sizeof(IOCTL_PARAMETERS_32))
    {
        status = STATUS_INVALID_PARAMETER;
    }
    else
    {
        LocalParam.Addr = Ptr32ToPtr(params32->Addr);
        LocalParam.Handle = Handle32ToHandle(params32->Handle);
        LocalParam.Length = params32->Length;

        /* Handle the IOCTL here. */
        ...

        status = STATUS_SUCCESS;
        Irp->IoStatus.Information = 0;
    }
}
else
{
    /* 64-bit process IOCTL */
    PIOCTL_PARAMETERS params;

    params = (PIOCTL_PARAMETERS)(Irp->AssociatedIrp.SystemBuffer);
    if (irpSp->Parameters.DeviceIoControl.InputBufferLength < sizeof(IOCTL_PARAMETERS))
    {
        status = STATUS_INVALID_PARAMETER;
    }
    else
    {
        RtlCopyMemory(&LocalParam, params, sizeof(IOCTL_PARAMETERS));

        /* Handle the IOCTL here. */
        ...

        status = STATUS_SUCCESS;
    }
    Irp->IoStatus.Information = 0;
}
```

Extended Example: Defining a "64Bit" Field

12/5/2018 • 2 minutes to read • Edit Online

The following example shows how to modify a 32-bit driver for 64-bit by adding a "64Bit" field to the IOCTL control code. Note that this example shows only the portions of the driver code that need to be modified.

Original Driver Code

The following is the 32-bit version of the driver:

Header File

```
#define REGISTER_FUNCTION 0     // Define the IOCTL function code

#define IOCTL_REGISTER   CTL_CODE(FILE_DEVICE_UNKNOWN, \
  REGISTER_FUNCTION, METHOD_BUFFERED, FILE_ANY_ACCESS)

typedef struct _IOCTL_PARAMETERS {
    PVOID   Addr;
    SIZE_T  Length;
    HANDLE  Handle;
} IOCTL_PARAMETERS, *PIOCTL_PARAMETERS;
```

DeviceControl Dispatch Routine

```
NTSTATUS
TestdrvDeviceControl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    PIO_STACK_LOCATION irpSp;
    NTSTATUS status;
    PIOCTL_PARAMETERS params;
    IOCTL_PARAMETERS  LocalParam;
    PIOCTL_PARAMETERS_32 params32;

    //
    // Get a pointer to the current parameters for this request. The
    // information is contained in the current stack location.
    //
    irpSp = IoGetCurrentIrpStackLocation(Irp);
    //
    // Case on the device control code
    //
    switch (irpSp->Parameters.DeviceIoControl.IoControlCode) {
    case IOCTL_REGISTER:
        params = (PIOCTL_PARAMETERS)
            (Irp->AssociatedIrp.SystemBuffer);
        if (irpSp->Parameters.DeviceIoControl.InputBufferLength <
               sizeof(IOCTL_PARAMETERS)) {
            status = STATUS_INVALID_PARAMETER;
        } else {
            RtlCopyMemory(&LocalParam,  params,
              sizeof(IOCTL_PARAMETERS));
            /* Handle the ioctl here */
            status = STATUS_SUCCESS;
        }
        Irp->IoStatus.Information = 0;
            break;
    //
    // Unrecognized device control request
    //
    default:
        Irp->IoStatus.Information = 0;
        status = STATUS_INVALID_PARAMETER;
        break;
    }
    //
    // If status is pending, mark the IRP pending and start the
    // request in a cancelable state. Otherwise, complete the IRP.
    //
    Irp->IoStatus.Status = status;
 IoCompleteRequest(Irp, IO_NO_INCREMENT);
 return(status);
}
```

Driver Code With Thunking Support

The following is the 64-bit version of the driver:

Header File

```
#define REGISTER_FUNCTION 0     // Define the IOCTL function code

#ifdef  _WIN64
#define CLIENT_64BIT   0x800
#define REGISTER_FUNCTION 0
#define IOCTL_REGISTER   CTL_CODE(FILE_DEVICE_UNKNOWN, \
  CLIENT_64BIT|REGISTER_FUNCTION, METHOD_BUFFERED, FILE_ANY_ACCESS)
#else
#define IOCTL_REGISTER   CTL_CODE(FILE_DEVICE_UNKNOWN, \
  REGISTER_FUNCTION, METHOD_BUFFERED, FILE_ANY_ACCESS)
#endif

typedef struct _IOCTL_PARAMETERS {
    PVOID   Addr;
    SIZE_T  Length;
    HANDLE  Handle;
} IOCTL_PARAMETERS, *PIOCTL_PARAMETERS;
```

DeviceControl Dispatch Routine

```
#ifdef _WIN64
#define IOCTL_REGISTER_32   CTL_CODE(FILE_DEVICE_UNKNOWN, \
  REGISTER_FUNCTION, METHOD_BUFFERED, FILE_ANY_ACCESS)
 #endif

...

#ifdef _WIN64
typedef struct _IOCTL_PARAMETERS_32 {
    VOID*POINTER_32  Addr;
    INT32            Length;
    VOID*POINTER_32  Handle;
} IOCTL_PARAMETERS_32, *PIOCTL_PARAMETERS_32;
 #endif

...

NTSTATUS
TestdrvDeviceControl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
    PIO_STACK_LOCATION irpSp;
    NTSTATUS status;
    PIOCTL_PARAMETERS params;
    IOCTL_PARAMETERS  LocalParam;
    PIOCTL_PARAMETERS_32 params32;

    //
    // Get a pointer to the current parameters for this request. The
    // information is contained in the current stack location.
    //
    irpSp = IoGetCurrentIrpStackLocation(Irp);
    //
    // Case on the device control code
    //
    switch (irpSp->Parameters.DeviceIoControl.IoControlCode) {
#ifdef  _WIN64
    case IOCTL_REGISTER_32:
        params32 = (PIOCTL_PARAMETERS_32)
          (Irp->AssociatedIrp.SystemBuffer);
        if (irpSp->Parameters.DeviceIoControl.InputBufferLength <
            sizeof(IOCTL_PARAMETERS_32)) {
            status = STATUS_INVALID_PARAMETER;
        } else {
```

```
            LocalParam.Addr = params32->Addr;
            LocalParam.Handle = params32->Handle;
            LocalParam.Length = params32->Length;
            /* Handle the ioctl here */
            status = STATUS_SUCCESS;
            Irp->IoStatus.Information = 0;
        }
        break;
 #endif
    case IOCTL_REGISTER:
        params = (PIOCTL_PARAMETERS)
            (Irp->AssociatedIrp.SystemBuffer);
        if (irpSp->Parameters.DeviceIoControl.InputBufferLength <
            sizeof(IOCTL_PARAMETERS)) {
            status = STATUS_INVALID_PARAMETER;
        } else {
            RtlCopyMemory(&LocalParam, params,
                sizeof(IOCTL_PARAMETERS));
            /* Handle the ioctl here */
            status = STATUS_SUCCESS;
        }
        Irp->IoStatus.Information = 0;
        break;
    //
    // Unrecognized device control request
    //
    default:
        Irp->IoStatus.Information = 0;
        status = STATUS_INVALID_PARAMETER;
        break;
    }
    //
    // If status is pending, mark the IRP pending and start the
    // request in a cancelable state. Otherwise, complete the IRP.
    //
    Irp->IoStatus.Status = status;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return(status);
}
```

Extended Example: Using IoIs32bitProcess

6/25/2019 • 2 minutes to read • Edit Online

The following example shows how to modify a 32-bit driver for 64-bit by adding a call to **IoIs32bitProcess**. Note that this example shows only the portions of the driver code that need to be modified.

Original Driver Code

```
typedef struct _TESTDRV_EVENT_BUFFER {
     HANDLE Handle;
     ULONG Key;
} TESTDRV_EVENT_BUFFER, *PTESTDRV_EVENT_BUFFER;

NTSTATUS
TestdrvFsControl (
    IN PTESTDRV_DEVICE_OBJECT TestdrvDeviceObject,
    IN PIRP Irp
    )
{

    ...

    InputBufferLength =
        IrpSp->Parameters.FileSystemControl.InputBufferLength;


    if (InputBufferLength < sizeof(TESTDRV_EVENT_BUFFER)) {

        DebugTrace(0, Dbg, "System buffer size is too small\n", 0);

        FsRtlCompleteRequest( Irp, STATUS_INVALID_PARAMETER );
        return STATUS_INVALID_PARAMETER;
    }

    Buffer = Irp->AssociatedIrp.SystemBuffer;

    // start using the event buffer

    ...

}
```

Driver Code With Thunking Support

```
typedef struct _TESTDRV_EVENT_BUFFER {
     HANDLE Handle;
     ULONG Key;
} TESTDRV_EVENT_BUFFER, *PTESTDRV_EVENT_BUFFER;

//
// Define a 32-bit thunking structure
//

 #if defined(_WIN64)
typedef struct _TESTDRV_EVENT_BUFFER32 {
     UINT32 Handle;
     ULONG Key;
} TESTDRV_EVENT_BUFFER32, *PTESTDRV_EVENT_BUFFER32;
#endif

//
```

```
//
// Intercept the input buffer as a 32-bit structure and thunk it to
 //    64-bit
NTSTATUS
TestdrvFsControl (
    IN PTESTDRV_DEVICE_OBJECT TestdrvDeviceObject,
    IN PIRP Irp
    )
{
    TESTDRV_EVENT_BUFFER LocalBuffer;

    ...

    InputBufferLength  =
             IrpSp->Parameters.FileSystemControl.InputBufferLength;

#if defined(_WIN64)
    if (IoIs32bitProcess(Irp)) {
        PTESTDRV_EVENT_BUFFER32 Buffer32;

        if (InputBufferLength < sizeof(TESTDRV_EVENT_BUFFER32)) {
            DebugTrace(0, Dbg, "Irp32 : System buffer size is too
                        small\n", 0);

            FsRtlCompleteRequest( Irp, STATUS_INVALID_PARAMETER );
            return STATUS_INVALID_PARAMETER;
        }
        Buffer = &LocalBuffer;
        Buffer32 = Irp->AssociatedIrp.SystemBuffer;
        Buffer->Handle = (HANDLE)Buffer32->Handle;
        Buffer->Key = Buffer32->Key;
    }
    else {
#endif
        if (InputBufferLength < sizeof(TESTDRV_EVENT_BUFFER)) {

            DebugTrace(0, Dbg, "System buffer size is too small\n", 0);

            FsRtlCompleteRequest( Irp, STATUS_INVALID_PARAMETER );
            return STATUS_INVALID_PARAMETER;
        }

        Buffer = Irp->AssociatedIrp.SystemBuffer;
#if defined(_WIN64)
    }
#endif

    // start using the Event Buffer

    ...

}
```

Avoiding Misalignment of Fixed-Precision Data Types

6/25/2019 • 3 minutes to read • Edit Online

Unfortunately, it is possible for a data type to have the same size, but different alignment requirements, for 32-bit and 64-bit programming. Thus not all IOCTL/FSCTL buffer misalignment problems can be avoided by changing pointer-precision data types to fixed-precision types. This means that kernel-mode driver IOCTLs and FSCTLs that pass buffers containing certain fixed-precision data types (or pointers to them) may also need to be thunked.

Which Data Types Are Affected

The problem affects fixed-precision data types that are themselves structures. This is because the rules for determining alignment requirements for structures are platform-specific.

For example, **__int64**, LARGE_INTEGER, and KFLOATING_SAVE must be aligned on a 4-byte boundary on x86 platforms. However, on Itanium-based machines, they must be aligned on an 8-byte boundary.

To determine the alignment requirement for a given data type on a particular platform, use the **TYPE_ALIGNMENT** macro on that platform.

How To Fix the Problem

In the following example, the IOCTL is a METHOD_NEITHER IOCTL, so the **Irp->UserBuffer** pointer is passed directly from the user-mode application to the kernel-mode driver. No validation is performed on buffers used in IOCTLs and FSCTLs. Thus a call to **ProbeForRead** or **ProbeForWrite** is required before the buffer pointer can be safely dereferenced.

Assuming that the 32-bit application has passed a valid value for **Irp->UserBuffer**, the LARGE_INTEGER structure pointed to by **p->DeviceTime** will be aligned on a 4-byte boundary. **ProbeForRead** checks this alignment against the value passed in its *Alignment* parameter, which in this case is **TYPE_ALIGNMENT** (LARGE_INTEGER). On x86 platforms, this macro expression returns 4 (bytes). However, on Itanium-based machines, it returns 8, causing **ProbeForRead** to raise a STATUS_DATATYPE_MISALIGNMENT exception.

Note Removing the **ProbeForRead** call does not fix the problem, but only makes it harder to diagnose.

```
typedef struct _IOCTL_PARAMETERS2 {
    LARGE_INTEGER DeviceTime;
} IOCTL_PARAMETERS2, *PIOCTL_PARAMETERS2;

#define SETTIME_FUNCTION 1
#define IOCTL_SETTIME CTL_CODE(FILE_DEVICE_UNKNOWN, \
            SETTIME_FUNCTION, METHOD_NEITHER, FILE_ANY_ACCESS)

...

case IOCTL_SETTIME:
    PIOCTL_PARAMETERS2 p = (PIOCTL_PARAMETERS2)Irp->UserBuffer;

    try {
        if (Irp->RequestorMode != KernelMode) {
            ProbeForRead ( p->DeviceTime,
                      sizeof( LARGE_INTEGER ),
                      TYPE_ALIGNMENT( LARGE_INTEGER ));
    }
    status = DoSomeWork(p->DeviceTime);

 } except( EXCEPTION_EXECUTE_HANDLER ) {
```

The following sections tell how to fix the problem described above. Note that all code snippets have been edited for brevity.

Solution 1: Copy the Buffer

The safest way to avoid misalignment problems is to make a copy of the buffer before accessing its contents, as in the following example.

```
case IOCTL_SETTIME: {
    PIOCTL_PARAMETERS2 p = (PIOCTL_PARAMETERS2)Irp->UserBuffer;
#if _WIN64
    IOCTL_PARAMETERS2 LocalParams2;

    RtlCopyMemory(&LocalParams2, p, sizeof(IOCTL_PARAMETERS2));
    p = &LocalParams2;
#endif

    status = DoSomeWork(p->DeviceTime);
    break;
}
```

This solution can be optimized for better performance by first checking whether the buffer contents are correctly aligned. If so, the buffer can be used as is. Otherwise, the driver makes a copy of the buffer.

```
case IOCTL_SETTIME: {
    PIOCTL_PARAMETERS2 p = (PIOCTL_PARAMETERS2)Irp->UserBuffer;
#if _WIN64
    IOCTL_PARAMETERS2 LocalParams2;

    if ( (ULONG_PTR)p & (TYPE_ALIGNMENT(IOCTL_PARAMETERS2)-1)) {
        // The buffer contents are not correctly aligned for this
        // platform, so copy them into a properly aligned local
        // buffer.
        RtlCopyMemory(&LocalParams2, p, sizeof(IOCTL_PARAMETERS2));
        p = &LocalParams2;
    }
#endif

    status = DoSomeWork(p->DeviceTime);
    break;
}
```

Solution 2: Use the UNALIGNED Macro

The **UNALIGNED** macro tells the C compiler to generate code that can access the **DeviceTime** field without taking an alignment fault. Note that using this macro on Itanium-based platforms is likely to make your driver significantly larger and slower.

```
typedef struct _IOCTL_PARAMETERS2 {
    LARGE_INTEGER DeviceTime;
} IOCTL_PARAMETERS2;
typedef IOCTL_PARAMETERS2 UNALIGNED *PIOCTL_PARAMETERS2;
```

Pointers Are Also Affected

The misalignment problem described earlier can also occur in buffered I/O requests. In the following example, the IOCTL buffer contains an embedded pointer to a LARGE_INTEGER structure.

```
typedef struct _IOCTL_PARAMETERS3 {
    LARGE_INTEGER *pDeviceCount;
} IOCTL_PARAMETERS3, *PIOCTL_PARAMETERS3;0

#define COUNT_FUNCTION 1
#define IOCTL_GETCOUNT CTL_CODE(FILE_DEVICE_UNKNOWN, \
            COUNT_FUNCTION, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

Like the METHOD_NEITHER IOCTL and FSCTL buffer pointers described earlier, pointers embedded in buffered I/O requests are also passed directly from the user-mode application to the kernel-mode driver. No validation is performed on these pointers. Thus a call to **ProbeForRead** or **ProbeForWrite**, enclosed in a **try/except** block, is required before the embedded pointer can be safely dereferenced.

As in the earlier example, assuming that the 32-bit application has passed a valid value for **pDeviceCount**, the LARGE_INTEGER structure pointed to by **pDeviceCount** will be aligned on a 4-byte boundary. **ProbeForRead** and **ProbeForWrite** check this alignment against the value of the *Alignment* parameter, which in this case is TYPE_ALIGNMENT (LARGE_INTEGER). On x86 platforms, this macro expression returns 4 (bytes). However, on Itanium-based machines, it returns 8, causing **ProbeForRead** or **ProbeForWrite** to raise a STATUS_DATATYPE_MISALIGNMENT exception.

This problem can be fixed either by making a properly aligned copy of the LARGE_INTEGER structure, as in Solution 1, or by using the UNALIGNED macro as follows:

```
typedef struct _IOCTL_PARAMETERS3 {
    LARGE_INTEGER UNALIGNED *pDeviceCount;
} IOCTL_PARAMETERS3, *PIOCTL_PARAMETERS3;
```

Driver x64 Restrictions

12/21/2018 • 2 minutes to read • Edit Online

On x64-based systems, kernel code and certain kernel data structures are protected from modification. Any driver that attempts to modify such code or data will cause the system to bug check (with the CRITICAL_STRUCTURE_CORRUPTION bug check).

Drivers for x64-based systems must avoid operations that might trigger this bug check. In particular, drivers must not:

- Attempt to modify kernel code at run time.
- Implement and use their own stacks.
- Modify hardware dispatch tables, such as the interrupt dispatch table (IDT) or global descriptor table (GDT).
- Modify undocumented kernel data structures.

Even though the preceding operations will not trigger a bug check on x86-based or Itanium-based systems, drivers should not perform any of these operations on any platform. These operations might not work in future versions of the Microsoft Windows operating system.

For more information about modifying kernel code and data structures, see the Patching Policy for x64-based Systems white paper and the 64-Bit Patching FAQ.

For general information about programming with a 64-bit compiler, see 64-Bit Programming with Visual C++.

Windows kernel obsolete macros

6/25/2019 • 2 minutes to read • Edit Online

This topic summarizes the following obsolete macros:

MACRO	DESCRIPTION
COMPUTE_PAGES_SPANNED	Use **ADDRESS_AND_SIZE_TO_SPAN_PAGES** instead.

Windows kernel obsolete routines

6/25/2019 • 2 minutes to read • Edit Online

The following obsolete routines are exported to support existing binaries:

OBSOLETE ROUTINE	DESCRIPTION
ExAcquireResourceExclusive	Use **ExAcquireResourceExclusiveLite** instead.
ExAcquireResourceShared	Use **ExAcquireResourceSharedLite** instead.
ExAllocateFromZone	Use lookaside lists instead. For more information, see Buffer Management.
ExConvertExclusiveToShared	Use **ExConvertExclusiveToSharedLite** instead.
ExDeleteResource	Use **ExDeleteResourceLite** instead.
ExExtendZone	Use lookaside lists instead. For more information, see Buffer Management.
ExFreeToZone	Use lookaside lists instead. For more information, see Buffer Management.
ExInitializeResource	Use **ExInitializeResourceLite** instead.
ExInitializeWorkItem	Use **IoAllocateWorkItem** instead.
ExInitializeZone	Use lookaside lists instead. For more information, see Buffer Management.
ExInterlockedAllocateFromZone	Use lookaside lists instead. For more information, see Buffer Management.
ExInterlockedDecrementLong	Use **InterlockedDecrement** instead.
ExInterlockedExchangeAddLargeInteger	For more information about atomically adding two 64-bit numbers, see InterlockedExchangeAdd64.
ExInterlockedExchangeUlong	Use **InterlockedExchange** instead.

OBSOLETE ROUTINE	DESCRIPTION
ExInterlockedExtendZone	Use lookaside lists instead. For more information, see Buffer Management.
ExInterlockedFreeToZone	Use lookaside lists instead. For more information, see Buffer Management.
ExInterlockedIncrementLong	Use **InterlockedIncrement** instead.
ExIsFullZone	Use lookaside lists instead. For more information, see Buffer Management.
ExIsObjectInFirstZoneSegment	Use lookaside lists instead. For more information, see Buffer Management.
ExIsResourceAcquired	Use **ExIsResourceAcquiredLite** instead.
ExIsResourceAcquiredExclusive	Use **ExIsResourceAcquiredExclusiveLite** instead.
ExIsResourceAcquiredShared	Use **ExIsResourceAcquiredSharedLite** instead.
ExReleaseResource	Use **ExReleaseResourceLite** instead.
ExReleaseResourceForThread	Use **ExReleaseResourceForThreadLite** instead.
IoAllocateAdapterChannel	Use **AllocateAdapterChannel** instead.
IoAssignResources	Drivers of PnP devices are assigned resources by the PnP manager, which passes resource lists with each **IRP_MN_START_DEVICE** request. Drivers that must support a legacy device that cannot be enumerated by the PnP manager should use **IoReportDetectedDevice** and **IoReportResourceForDetection** instead.
IoAttachDeviceByPointer	Use **IoAttachDeviceToDeviceStack** instead.
IoFlushAdapterBuffers	Use **FlushAdapterBuffers** instead.
IoFreeAdapterChannel	Use **FreeAdapterChannel** instead.
IoFreeMapRegisters	Use **FreeMapRegisters** instead.

OBSOLETE ROUTINE	DESCRIPTION
IoMapTransfer	Use **MapTransfer** instead.
IoQueryDeviceDescription	This routine retrieves hardware configuration information about a given bus, controller or peripheral object, or any combination of these three types from the **\Registry\Machine\Hardware\Description** tree. Drivers that require hardware configuration information should use **IoGetDeviceProperty** instead.
IoReportResourceUsage	This routine claims hardware resources, such as an interrupt vector, device memory range or a particular DMA controller channel in the **\Registry\Machine\Hardware\ResourceMap** tree, so that a subsequently loaded driver cannot attempt to use the same resources. If a new driver must support a legacy device that is not PnP-enumerable, the driver should call **IoReportResourceForDetection** to claim resources for the device.
KeGetDcacheFillSize	Drivers should call **GetDmaAlignment** instead.
MmCreateMdl	Use **IoAllocateMdl** instead.
MmIsNonPagedSystemAddressValid	

Related topics

AllocateAdapterChannel
Buffer Management
ExAcquireResourceExclusiveLite
ExAcquireResourceSharedLite
ExConvertExclusiveToSharedLite
ExDeleteResourceLite
ExInitializeResourceLite
ExIsResourceAcquiredExclusiveLite
ExIsResourceAcquiredSharedLite
ExReleaseResourceForThreadLite
ExReleaseResourceLite
InterlockedDecrement
InterlockedExchange
InterlockedIncrement
FlushAdapterBuffers
FreeAdapterChannel
FreeMapRegisters
GetDmaAlignment
InterlockedExchangeAdd64
IoAllocateMdl
IoAllocateWorkItem
IoAttachDeviceToDeviceStack

IoGetDeviceProperty
IoReportDetectedDevice
IoReportResourceForDetection
IRP_MN_START_DEVICE
MapTransfer

Windows kernel routines reserved for system use

6/25/2019 • 2 minutes to read • Edit Online

The following routines are reserved for system use:

ROUTINE	DESCRIPTION
IoAcquireRemoveLockEx	See **IoAcquireRemoveLock**.
IoInitializeRemoveLockEx	Use **IoInitializeRemoveLock** instead.
IoReleaseRemoveLockAndWaitEx	See **IoReleaseRemoveLockAndWait**.
IoReleaseRemoveLockEx	See **IoReleaseRemoveLock**.

Related topics

IoAcquireRemoveLock
IoInitializeRemoveLock
IoReleaseRemoveLock
IoReleaseRemoveLockAndWait

Windows kernel run-time library obsolete routines

6/25/2019 • 2 minutes to read • Edit Online

The following run-time library obsolete routines are exported to support existing driver binaries:

OBSOLETE ROUTINE	DESCRIPTION
RtlEnlargedIntegerMultiply	For better performance, use the **RtlLongMult** routine if the result will fit into a 32-bit signed integer. Otherwise, use the compiler support for 64-bit integer operations.
RtlEnlargedUnsignedDivide	For better performance, use the compiler support for 64-bit integer operations.
RtlEnlargedUnsignedMultiply	For better performance, use the **RtlULongMult** routine if the result will fit into a 32-bit unsigned integer. Otherwise, use the compiler support for 64-bit integer operations.
RtlExtendedIntegerMultiply	For better performance, use the compiler support for 64-bit integer operations.
RtlExtendedLargeIntegerDivide	For better performance, use the compiler support for 64-bit integer operations.
RtlExtendedMagicDivide	For better performance, use the compiler support for 64-bit integer operations.
RtlFillBytes	Fills a caller-supplied buffer with the given unsigned character. Use **RtlFillMemory** instead.
RtlLargeIntegerAdd	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerAnd	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerArithmeticShift	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerDivide	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerEqualTo	For better performance, use the compiler support for 64-bit integer operations.

OBSOLETE ROUTINE	DESCRIPTION
RtlLargeIntegerEqualToZero	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerGreaterOrEqualToZero	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerGreaterThan	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerGreaterThanOrEqualTo	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerGreaterThanZero	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerLessOrEqualToZero	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerLessThan	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerLessThanOrEqualTo	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerLessThanZero	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerNegate	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerNotEqualTo	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerNotEqualToZero	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerShiftLeft	For better performance, use the compiler support for 64-bit integer operations.
RtlLargeIntegerShiftRight	For better performance, use the compiler support for 64-bit integer operations.

OBSOLETE ROUTINE	DESCRIPTION
RtlLargeIntegerSubtract	For better performance, use the compiler support for 64-bit integer operations.
RtlZeroBytes	Fills a block of memory with zeros, given a pointer to the block and the length, in bytes, to be filled. For better performance, use **RtlZeroMemory**.

Related topics

RtlFillMemory
RtlLongMult
RtlZeroMemory

Windows kernel global variables

1/10/2019 • 2 minutes to read • Edit Online

Kernel global variables.

VARIABLE	DECLARATION	DESCRIPTION
Mm64BitPhysicalAddress	`PBOOLEAN Mm64BitPhysicalAddress` Declared in Wdm.h	Specifies whether the hardware and operating system support 64-bit physical addresses. Points to a value that is **TRUE** if the hardware and operating system support 64-bit physical addresses, and is **FALSE** otherwise. For more information about how to use this variable in your driver, see Performing DMA in 64-Bit Windows.
MmBadPointer	`PVOID MmBadPointer;` Declared in Wdm.h	A pointer to a memory location that is guaranteed to be invalid. **Note** Starting with Windows 8.1, **MmBadPointer** is deprecated. Drivers should use the **MM_BAD_POINTER** macro instead. The operating system generates a bug check if the memory address that is specified by the **MmBadPointer** variable is accessed. You can use **MmBadPointer** to debug your driver code. Set any uninitialized pointer variables to **MmBadPointer** to find the first time that your code tries to dereference an invalid pointer. All addresses within PAGE_SIZE of **MmBadPointer** are guaranteed to be invalid. For example, if *Address* is a pointer and if **MmBadPointer** <= *Address* < **MmBadPointer** + PAGE_SIZE, attempts to access **Address* causes the operating system to generate a bug check. **MmBadPointer** + PAGE_SIZE is not guaranteed to be invalid.
PsInitialSystemProcess	`PEPROCESS PsInitialSystemProcess;` Declared in Ntddk.h	Points to the **EPROCESS** structure for the system process.

VARIABLE	DECLARATION	DESCRIPTION
NLS_MB_CODE_PAGE_TAG	`extern BOOLEAN NLS_MB_CODE_PAGE_TAG;`	Specifies whether a code page is a single-byte or multibyte code page. **NLS_MB_CODE_PAGE_TAG** is **TRUE** for multibyte code pages and **FALSE** for single-byte code pages. NLS_MB_CODE_PAGE_TAG is reserved for system use. From user mode, call GetCPInfoEx instead. When possible, your application should use Unicode instead of code pages.

Related topics

EPROCESS

GetCPInfoEx

MM_BAD_POINTER

Performing DMA in 64-Bit Windows

Windows kernel macros

10/7/2019 • 25 minutes to read • Edit Online

The following list contains Windows kernel macros:

ADDRESS_AND_SIZE_TO_SPAN_PAGES

Defined in: Wdm.h

The **ADDRESS_AND_SIZE_TO_SPAN_PAGES** macro returns the number of pages spanned by the virtual range defined by a virtual address and the size in bytes of a transfer request.

Va [in]

PVOID

Pointer to the virtual address that is the base of the range.

Size [in]

ULONG

Specifies the size in bytes of the transfer request.

Return value

ULONG

ADDRESS_AND_SIZE_TO_SPAN_PAGES returns the number of pages spanned by the virtual range starting at *Va*.

Drivers that make DMA transfers call **ADDRESS_AND_SIZE_TO_SPAN_PAGES** to determine whether a transfer request must be split into a sequence of device DMA operations.

A driver can use the system-defined constant PAGE_SIZE to determine whether the number of bytes to be transferred is less than the virtual memory page size of the current platform.

Callers of **ADDRESS_AND_SIZE_TO_SPAN_PAGES** can be running at any IRQL. The caller must ensure that the specified parameters do not cause memory overflow.

Available starting with Windows 2000.

BYTE_OFFSET

Defined in: Wdm.h

The **BYTE_OFFSET** macro takes a virtual address and returns the byte offset of that address within the page.

Va [in]

PVOID

Pointer to the virtual address.

Return value

ULONG

BYTE_OFFSET returns the offset portion of the virtual address.

Available starting with Windows 2000.

IRQL: Any level

BYTES_TO_PAGES

Defined in: Wdm.h

The **BYTES_TO_PAGES** macro takes the size in bytes of the transfer request and calculates the number of pages required to contain the bytes.

Size [in]

ULONG

Specifies the size in bytes of the transfer request.

Return value

ULONG

BYTES_TO_PAGES returns the number of pages required to contain the specified number of bytes.

The system-defined constant PAGE_SIZE can be used to determine whether a given length in bytes for a transfer is less than the virtual memory page size of the current platform.

Available starting with Windows 2000.

IRQL: Any level

CONTAINING_RECORD

Defined in: Ntdef.h

The **CONTAINING_RECORD** macro returns the base address of an instance of a structure given the type of the structure and the address of a field within the containing structure.

Address [in]

PCHAR

A pointer to a field in an instance of a structure of type *Type*.

Type [in]

TYPE

The name of the type of the structure whose base address is to be returned.

Field [in]

PCHAR

The name of the field pointed to by *Address* and which is contained in a structure of type *Type*.

Return value

PCHAR

Returns the address of the base of the structure containing *Field*.

Called to determine the base address of a structure whose type is known when the caller has a pointer to a field

inside such a structure. This macro is useful for symbolically accessing other fields in a structure of known type.

Available starting with Windows 2000.

IRQL: Any level

IoSkipCurrentIrpStackLocation

Defined in: Wdm.h

\nThe **IoSkipCurrentIrpStackLocation** macro modifies the system's **IO_STACK_LOCATION** array pointer, so that when the current driver calls the next-lower driver, that driver receives the same **IO_STACK_LOCATION** structure that the current driver received.

Irp [in, out]

PIRP

A pointer to the IRP.

Return value

VOID

When your driver sends an IRP to the next-lower driver, your driver can call **IoSkipCurrentIrpStackLocation** if you do not intend to provide an *IoCompletion* routine (the address of which is stored in the driver's **IO_STACK_LOCATION** structure). If you call **IoSkipCurrentIrpStackLocation** before calling **IoCallDriver**, the next-lower driver receives the same **IO_STACK_LOCATION** that your driver received.

If you intend to provide an *IoCompletion* routine for the IRP, your driver should call **IoCopyCurrentIrpStackLocationToNext** instead of **IoSkipCurrentIrpStackLocation**. If a badly written driver makes the mistake of calling **IoSkipCurrentIrpStackLocation** and then setting a completion routine, this driver might overwrite a completion routine set by the driver below it.

If the driver has pended an IRP, the driver should not be calling **IoSkipCurrentIrpStackLocation** before it passes the IRP to the next lower driver. If the driver calls **IoSkipCurrentIrpStackLocation** on a pended IRP before passing it to the next lower driver, the SL_PENDING_RETURNED flag is still set in the **Control** member of the I/O stack location for the next driver. Because the next driver owns that stack location and might modify it, it could potentially clear the pending flag. This situation might cause the operating system to issue a bug check or the processing of the IRP to never be completed.

Instead, a driver that has pended an IRP should call **IoCopyCurrentIrpStackLocationToNext** to set up a new stack location for the next lower driver before it calls **IoCallDriver**.

If your driver calls **IoSkipCurrentIrpStackLocation**, be careful not to modify the **IO_STACK_LOCATION** structure in a way that could unintentionally affect the lower driver or the system's behavior with respect to that driver. Examples include modifying the **IO_STACK_LOCATION** structure's **Parameters** union or calling **IoMarkIrpPending**.

Available starting with Windows 2000.

IRQL: Any level

KeInitializeCallbackRecord

Defined in: Wdm.h

The **KeInitializeCallbackRecord** macro initializes a **KBUGCHECK_CALLBACK_RECORD** or **KBUGCHECK_REASON_CALLBACK_RECORD** structure.

CallbackRecord [in]

PKBUGCHECK_CALLBACK_RECORD

Pointer to either a **KBUGCHECK_CALLBACK_RECORD** or a **KBUGCHECK_REASON_CALLBACK_RECORD** structure. The structure must be in resident memory, such as nonpaged pool.

Return value

VOID

Available in Windows 2000 and later versions of Windows.

IRQL: Any level

MM_BAD_POINTER

Defined in: Wdm.h

Your driver can use the **MM_BAD_POINTER** macro as a bad pointer value to assign to a pointer variable that is either uninitialized or no longer valid. An attempt to access the memory location pointed to by this invalid pointer variable will cause a bug check.

On many hardware platforms, address 0 (frequently represented as named constant **NULL**) is an invalid address, but driver developers should not assume that address 0 is universally invalid across all platforms. Setting uninitialized or invalid pointer variables to address 0 might not always guarantee that inappropriate accesses through these pointers will be detected.

In contrast, the value **MM_BAD_POINTER** is guaranteed to be an invalid address on every platform on which a driver runs.

On platforms on which address 0 is an invalid address, a driver that accesses address 0 at IRQL < DISPATCH_LEVEL causes an exception (access violation) that can be inadvertently caught by a `try/except` statement. Thus, the driver's exception handling code might mask the invalid access and prevent it from being detected during debugging. However, an access of the **MM_BAD_POINTER** address is guaranteed to cause a bug check, which cannot be masked by an exception handler.

The following code example shows how to assign the value **MM_BAD_POINTER** to a pointer variable named `ptr`. The Ntdef.h header file defines the PUCHAR type to be a pointer to an `unsigned char`.

```
PUCHAR ptr = (PUCHAR)MM_BAD_POINTER; // Now _ptr is guaranteed to fault._
```

After `ptr` is set to **MM_BAD_POINTER**, an attempt to access the memory location pointed to by `ptr` will cause a bug check.

In fact, **MM_BAD_POINTER** is the base address of an entire page of invalid addresses. Therefore, any access of an address in the range **MM_BAD_POINTER** to (**MM_BAD_POINTER** + **PAGE_SIZE** - 1) will cause a bug check.

Starting with Windows 8.1, the **MM_BAD_POINTER** macro is defined in the Wdm.h header file. However, driver code that uses this macro definition can run in previous versions of Windows starting with Windows Vista.

Starting with Windows Vista, the **MmBadPointer** global variable is available as a pointer to a pointer value that is guaranteed to be an invalid address. However, starting with Windows 8.1, the use of **MmBadPointer** is deprecated, and you should update your drivers to use the **MM_BAD_POINTER** macro instead.

Available starting with Windows 8.1. Compatible with previous versions of Windows starting with Windows Vista._

MmGetMdlByteCount

Defined in: Wdm.h

The **MmGetMdlByteCount** macro returns the length, in bytes, of the buffer described by the specified MDL.

Mdl [in]

PMDL

A pointer to an **MDL** structure that describes the layout of a virtual memory buffer in physical memory. For more information, see Using MDLs.

Return value

ULONG

MmGetMdlByteCount returns the length, in bytes, of the buffer described by *Mdl*.

Callers of **MmGetMdlByteCount** can be running at any IRQL. Usually, callers are running at IRQL <= DISPATCH_LEVEL.

Available starting with Windows 2000.

IRQL: Any level

MmGetMdlByteOffset

Defined in: Wdm.h

The **MmGetMdlByteOffset** macro returns the byte offset within the initial page of the buffer described by the given MDL.

Mdl [in]

PMDL

Pointer to an MDL.

Return value

ULONG

MmGetMdlByteOffset returns the offset in bytes.

Callers of **MmGetMdlByteOffset** can be running at any IRQL. Usually, callers are running at IRQL <= DISPATCH_LEVEL.

Available in Windows 2000 and later versions of Windows.

IRQL: Any level

MmGetMdlPfnArray

Defined in: Wdm.h

The **MmGetMdlPfnArray** macro returns a pointer to the beginning of the array of physical page numbers that are associated with a memory descriptor list (MDL).

Mdl [in]

PMDL

A pointer to an MDL.

Return value

PPFN_NUMBER

A pointer to the beginning of the array of physical page numbers associated with the MDL. The number of entries in the array is **ADDRESS_AND_SIZE_TO_SPAN_PAGES**(**MmGetMdlVirtualAddress**(*Mdl*), **MmGetMdlByteCount**(*Mdl*)). Each array element is an integer value of type PFN_NUMBER, which is Defined in: Wdm.h as follows:

```
cpp typedef ULONG PFN_NUMBER, *PPFN_NUMBER;
```

Note Changing the contents of the array can cause subtle system problems that are difficult to diagnose. We recommend that you do not read or change the contents of this array.

For pageable memory, the contents of the array are valid only for a buffer locked with **MmProbeAndLockPages**. For nonpaged pool, the contents of the array are valid only for an MDL updated with **MmBuildMdlForNonPagedPool**, **MmAllocatePagesForMdlEx**, or **MmAllocatePagesForMdl**.

For more information about MDLs, see Using MDLs.

Available starting with Windows 2000.

IRQL: Any level

MmGetMdlVirtualAddress

Defined in: Wdm.h

The **MmGetMdlVirtualAddress** macro returns the base virtual address of a buffer described by an MDL.

Mdl [in]

PMDL

Pointer to an MDL that describes the buffer for which to return the initial virtual address.

Return value

PVOID

MmGetMdlVirtualAddress returns the starting virtual address of the MDL.

MmGetMdlVirtualAddress returns a virtual address that is not necessarily valid in the current thread context. Lower-level drivers should not attempt to use the returned virtual address to access memory, particularly user memory space.

The returned address, used as an index to a physical address entry in the MDL, can be input to **MapTransfer**.

Callers of **MmGetMdlVirtualAddress** can be running at any IRQL. Usually, the caller is running at IRQL = DISPATCH_LEVEL because this routine is commonly called to obtain the *CurrentVa* parameter to **MapTransfer**.

Available in Windows 2000 and later versions of Windows.

IRQL: Any level

MmGetSystemAddressForMdlSafe

Defined in: Wdm.h

The **MmGetSystemAddressForMdlSafe** macro returns a nonpaged system-space virtual address for the buffer that the specified MDL describes.

Mdl [in]

PMDL

Pointer to a buffer whose corresponding base virtual address is to be mapped.

Priority [in]

Mm_PAGE_PRIORITY

Specifies an **MM_PAGE_PRIORITY** value that indicates the importance of success under low available PTE conditions. Specify a priority value of **LowPagePriority**, **NormalPagePriority**, or **HighPagePriority**. Starting with Windows 8, the specified priority value can be bitwise-ORed with the **MdlMappingNoWrite** or **MdlMappingNoExecute** flags.

- **LowPagePriority** indicates that the mapping request can fail if the system is fairly low on resources. An example of this situation is a noncritical network connection where the driver can handle the mapping failure.
- **NormalPagePriority** indicates that the mapping request can fail if the system is very low on resources. An example of this situation is a noncritical local file system request.
- **HighPagePriority** indicates that the mapping request must not fail unless the system is completely out of resources. An example of this situation is the paging file path in a driver.
- **MdlMappingNoWrite** indicates that the mapped physical pages are to be configured as no-write (read only) memory. Starting with Windows 8, this flag bit can be bitwise-ORed with the **MM_PAGE_PRIORITY** value to specify memory in which writes are disabled.
- **MdlMappingNoExecute** indicates that the mapped physical pages are to be configured as no-execute memory. Starting with Windows 8, this flag bit can be bitwise-ORed with the **MM_PAGE_PRIORITY** value to specify memory in which instruction execution is disabled. As a best practice, drivers written for Windows 8 and later versions of Windows should always specify no-execute memory unless executable memory is explicitly required.
- **Return value**

  PVOID

  MmGetSystemAddressForMdlSafe returns the base system-space virtual address that maps the physical pages that the specified MDL describes. If the pages are not already mapped to system address space and the attempt to map them fails, **NULL** is returned.

This routine maps the physical pages that are described by the specified MDL into system address space, if they are not already mapped to system address space.

Drivers of programmed-I/O (PIO) devices call this routine to map a user-mode buffer, which is described by the MDL at **Irp->MdlAddress** and which is already mapped to a user-mode virtual address range, to a range in system address space.

On entry to this routine, the specified MDL must describe physical pages that are locked down. A locked-down MDL can be built by using the **MmProbeAndLockPages**, **MmBuildMdlForNonPagedPool**, **IoBuildPartialMdl**, or **MmAllocatePagesForMdlEx** routine.

When the system-address-space mapping that is returned by **MmGetSystemAddressForMdlSafe** is no longer

needed, it must be released. The steps that are required to release the mapping depend on how the MDL was built. These are the four possible cases:

- If the MDL was built by a call to the **MmProbeAndLockPages** routine, it is not necessary to explicitly release the system-address-space mapping. Instead, a call to the **MmUnlockPages** routine releases the mapping, if one was allocated.
- If the MDL was built by a call to the **MmBuildMdlForNonPagedPool** routine, **MmGetSystemAddressForMdlSafe** reuses the existing system-address-space mapping instead of creating a new one. In this case, no cleanup is required (that is, unlocking and unmapping are not necessary).
- If the MDL was built by a call to the **IoBuildPartialMdl** routine, the driver must call either the **MmPrepareMdlForReuse** routine or the **IoFreeMdl** routine to release the system-address-space mapping.
- If the MDL was built by a call to the **MmAllocatePagesForMdlEx** routine, the driver must call the **MmUnmapLockedPages** routine to release the system-address-space mapping. If **MmGetSystemAddressForMdlSafe** is called more than one time for an MDL, subsequent **MmGetSystemAddressForMdlSafe** calls simply return the mapping that was created by the first call. One call to **MmUnmapLockedPages** is sufficient to release this mapping.

Starting with Windows 7 and Windows Server 2008 R2, it is not necessary to explicitly call **MmUnmapLockedPages** for an MDL that was created by **MmAllocatePagesForMdlEx**. Instead, a call to the **MmFreePagesFromMdl** routine releases the system-address-space mapping, if one was allocated.

To create a new system-address-space mapping, **MmGetSystemAddressForMdlSafe** calls **MmMapLockedPagesSpecifyCache** with the *CacheType* parameter set to **MmCached**. A driver that requires a cache type other than **MmCached** should call **MmMapLockedPagesSpecifyCache** directly instead of calling **MmGetSystemAddressForMdlSafe**. For more information about the *CacheType* parameter, see **MmMapLockedPagesSpecifyCache**.

In a call to **MmMapLockedPagesSpecifyCache**, the specified cache type is used only if the pages that are described by the MDL do not already have a cache type associated with them. However, in nearly all cases, the pages already have an associated cache type, and this cache type is used by the new mapping. An exception to this rule is for pages that are allocated by **MmAllocatePagesForMdl**, which sets the cache type to **MmCached** regardless of the original cache type of the pages.

Only one thread at a time can safely call **MmGetSystemAddressForMdlSafe** for a particular MDL because this routine assumes that the calling thread owns the MDL. However, **MmGetSystemAddressForMdlSafe** can be called more than one time for the same MDL either by making all calls from the same thread or, if the calls are from multiple threads, by explicitly synchronizing the calls.

If a driver must split a request into smaller requests, the driver can allocate additional MDLs, or the driver can use the **IoBuildPartialMdl** routine.

The returned base address has the same offset as the virtual address in the MDL.

Windows 98 does not support **MmGetSystemAddressForMdlSafe**. Use **MmGetSystemAddressForMdl** instead.

Because this macro calls **MmMapLockedPagesSpecifyCache**, using it may require linking to NtosKrnl.lib.

Available starting with Windows 2000.

IRQL <= DISPATCH_LEVEL

MmInitializeMdl

Defined in: Wdm.h

The **MmInitializeMdl** macro initializes the header of an MDL.

MemoryDescriptorList [in]

PMDL

A pointer to the buffer to initialize as an MDL. For more information, see the following section.

BaseVa [in]

PVOID

A pointer to the base virtual address of a buffer.

Length [in]

SIZE_T

Specifies the length, in bytes, of the buffer to be described by the MDL. This routine supports a maximum buffer length of MAXULONG bytes.

Return value

VOID

The buffer that *MemoryDescriptorList* points to must be allocated in nonpaged memory. The size, in bytes, of this buffer must be at least **sizeof**(MDL) + **sizeof**(PFN_NUMBER) * **ADDRESS_AND_SIZE_TO_SPAN_PAGES**(*BaseVa*, *Length*).

Available in Windows 2000 and later versions of Windows.

IRQL <= DISPATCH_LEVEL

MmPrepareMdlForReuse

Defined in: Wdm.h

The **MmPrepareMdlForReuse** macro releases the resources that are associated with a partial MDL so that the MDL can be reused.

Mdl [in]

PMDL

A pointer to a partial MDL that is to be prepared for reuse.

Return value

VOID

This macro is used by drivers that repeatedly use the same allocated MDL for the *TargetMdl* parameter in calls to the **IoBuildPartialMdl** routine. If, in a call to **MmPrepareMdlForReuse**, the specified partial MDL has an associated mapping to system address space, **MmPrepareMdlForReuse** releases the mapping so that the MDL can be reused.

MmPrepareMdlForReuse accepts only partial MDLs that are built by **IoBuildPartialMdl**. If **MmPrepareMdlForReuse** receives an MDL that is mapped to the system address space but was not built by **IoBuildPartialMdl**, **MmPrepareMdlForReuse** does not release the mapping, and, in checked builds, causes an assertion to fail.

For more information about partial MDLs, see Using MDLs.

Available in Windows 2000 and later versions of Windows.

IRQL <= DISPATCH_LEVEL

PAGE_ALIGN

Defined in: Wdm.h

The **PAGE_ALIGN** macro returns a page-aligned virtual address for a given virtual address.

Va [in]

PVOID

Pointer to the virtual address.

Return value

PVOID

PAGE_ALIGN returns a pointer to the page-aligned virtual address.

Available starting with Windows 2000.

IRQL: Any level

PAGED_CODE

Defined in: Wdm.h

The **PAGED_CODE** macro ensures that the calling thread is running at an IRQL that is low enough to permit paging.

Return value

VOID

If the IRQL > APC_LEVEL, the **PAGED_CODE** macro causes the system to ASSERT.

A call to this macro should be made at the beginning of every driver routine that either contains pageable code or accesses pageable code.

The **PAGED_CODE** macro checks the IRQL only at the point at which the driver code executes the macro. If the code subsequently raises the IRQL, the macro will not detect this change. Driver developers should use Static Driver Verifier and Driver Verifier to detect when the IRQL is raised improperly during the execution of a driver routine.

The **PAGED_CODE** macro works only in checked builds.

Available starting with Windows 2000.

PAGED_CODE_LOCKED

Defined in: Wdm.h

The **PAGED_CODE_LOCKED** macro asserts that the currently running code section is pageable and must have been locked into memory before it was run.

Return value

VOID

Pageable code must obey certain restrictions (such as IRQL <= APC_LEVEL), unless it is locked into place. A pageable routine that must be locked into place to work correctly should begin with a call to **PAGED_CODE_LOCKED**.

For more information about locking a code section into place, see Locking Pageable Code or Data.

PoSetDeviceBusy

Defined in: Wdm.h

The **PoSetDeviceBusy** macro notifies the power manager that the device associated with *IdlePointer* is busy.

IdlePointer [in, out]

PULONG

Specifies a non-**NULL** idle pointer that was previously returned by **PoRegisterDeviceForIdleDetection**. Note that **PoRegisterDeviceForIdleDetection** might return a **NULL** pointer. A caller of **PoSetDeviceBusy** must verify that the pointer is non-**NULL** before passing it to **PoSetDeviceBusy**.

Return value

VOID

Note The **PoSetDeviceBusyEx** routine is a direct replacement for the **PoSetDeviceBusy** macro. If you are writing new driver code for Windows Vista with Service Pack 1 (SP1) and later versions of Windows, call **PoSetDeviceBusyEx** instead of **PoSetDeviceBusy**.

A driver uses **PoSetDeviceBusy** along with **PoRegisterDeviceForIdleDetection** to enable system idle detection for its device. If a device that is registered for idle detection becomes idle, the power manager sends an **IRP_MN_SET_POWER** request to put the device in a requested sleep state.

PoSetDeviceBusy reports that the device is busy, so that the power manager can restart its idle countdown. If the device is not powered up, **PoSetDeviceBusy** does not change its state. That is, it does not cause the system to send a power-on request.

A driver should call **PoSetDeviceBusy** on every I/O request.

Available starting with Windows 2000.

IRQL: Any level

PsGetCurrentProcess

Defined in: Ntddk.h

Returns a pointer to the process of the current thread.

Return value

A pointer to an opaque process object.

Available starting with Windows 2000.

IRQL: Any level

READ_REGISTER_BUFFER_ULONG64

Defined in: Wdm.h

The **READ_REGISTER_BUFFER_ULONG64** macro reads a number of ULONG64 values from the specified

register address into a buffer.

Register [in]

PULONG64

Pointer to the register, which must be a mapped range in memory space.

Buffer [out]

PULONG64

Pointer to a buffer that an array of ULONG64 values is read into.

Count [in]

ULONG

Specifies the number of ULONG64 values to be read into the buffer.

Return value

VOID

The size of the *Buffer* buffer must be large enough to contain at least the specified number of ULONG64 values.

Callers of the **READ_REGISTER_BUFFER_ULONG64** macro can be running at any IRQL, assuming that the *Buffer* buffer is resident and the *Register* register is resident, mapped device memory.

Available only in 64-bit versions of Windows.

IRQL: Any level

READ_REGISTER_ULONG64

Defined in: Wdm.h

The **READ_REGISTER_ULONG64** macro reads a ULONG64 value from the specified register address.

*volatile *Register [in]*

ULONG64

Pointer to the register address, which must be a mapped range in memory space.

Return value

ULONG64

READ_REGISTER_ULONG64 returns the ULONG64 value that is read from the specified register address.

Callers of the **READ_REGISTER_ULONG64** macro can be running at any IRQL, assuming the *Register* address is resident, mapped device memory.

Available only in 64-bit versions of Windows.

IRQL: Any level

ROUND_TO_PAGES

Defined in: Wdm.h

The **ROUND_TO_PAGES** macro takes a size in bytes and rounds it up to the next full page.

Size [in]

ULONG_PTR

Specifies the size in bytes to round up to a page multiple.

Return value

ULONG_PTR

ROUND_TO_PAGES returns the input size rounded up to a multiple of the virtual memory page size for the current platform.

Callers of **ROUND_TO_PAGES** can be running at any IRQL. The caller must ensure that the supplied parameter cannot cause memory overflow.

IRQL: Any level

RtlEqualLuid

Defined in: Wdm.h

Return value

Available starting with Windows 2000.

IRQL: Any level

RtlInitEmptyAnsiString

Defined in: Wdm.h

The **RtlInitEmptyAnsiString** macro initializes an empty counted ANSI string.

DestinationString [out]

PANSI_STRING

Pointer to the **ANSI_STRING** structure to be initialized.

Buffer [in]

PCHAR

Pointer to a caller-allocated buffer to be used to contain a WCHAR string.

BufferSize [in]

USHORT

Length, in bytes, of the buffer that *Buffer* points to.

Return value

VOID

The members of the structure that the *DestinationString* parameter points to are initialized as follows.

- **Length**. Zero.
- **MaximumLength**. *BufferSize*.
- **Buffer**. *SourceString*.

To initialize a non-empty counted Unicode string, call **RtlInitAnsiString**.

Available in Microsoft Windows XP and later versions of Windows.

IRQL: Any level

RtlInitEmptyUnicodeString

Defined in: Wdm.h

The **RtlInitEmptyUnicodeString** macro initializes an empty counted Unicode string.

DestinationString [out]

PUNICODE_STRING

Pointer to the **UNICODE_STRING** structure to be initialized.

Buffer [in]

PWCHAR

Pointer to a caller-allocated buffer to be used to contain a WCHAR string.

BufferSize [in]

USHORT

Length, in bytes, of the buffer that *Buffer* points to.

Return value

VOID

The members of the structure that the *DestinationString* parameters points to are initialized as follows.

- **Length**. Zero.
- **MaximumLength**. *BufferSize*.
- **Buffer**. *SourceString*.

To initialize a non-empty counted Unicode string, call **RtlInitUnicodeString**.

Available starting with Windows XP.

IRQL: Any level

RtlIsZeroLuid

Defined in: Ntddk.h

The **RtlIsZeroLuid** macro determines if the specified LUID is the zero LUID.

L1 [in]

PLUID

Specifies the **LUID** to check.

Return value

BOOLEAN

RtlIsZeroLuid returns **TRUE** if *L1* is zero, and returns **FALSE** otherwise.

IRQL: Any level

RtlRetrieveUlong

Defined in: Wdm.h

The **RtlRetrieveUlong** macro retrieves a ULONG value from the source address, avoiding alignment faults. The destination address is assumed to be aligned.

DestinationAddress [out]

PULONG

Pointer to a ULONG-aligned location in which to store the ULONG value.

SourceAddress [in]

PULONG

Pointer to a location from which to retrieve the ULONG value.

Return value

VOID

Callers of **RtlRetrieveUlong** can be running at any IRQL if the given addresses are in nonpaged pool. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Available in Windows 2000 and later versions of Windows.

RtlRetrieveUshort

Defined in: Wdm.h

The **RtlRetrieveUshort** macro retrieves a USHORT value from the source address, avoiding alignment faults.

DestinationAddress [out]

PUSHORT

Pointer to a USHORT-aligned location in which to store the value.

SourceAddress [in]

PUSHORT

Pointer to a location from which to retrieve the value.

Return value

VOID

Callers of **RtlRetrieveUshort** can be running at any IRQL if the given addresses are in nonpaged pool. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Available in Windows 2000 and later versions of Windows.

IRQL: Any level

RtlStoreUlong

Defined in: Wdm.h

The **RtlStoreUlong** macro stores a ULONG value at a particular address, avoiding alignment faults.

Address [out]

PULONG

A pointer to a location in which to store the specified ULONG value.

Value [in]

ULONG

Specifies a ULONG value to be stored.

Return value

VOID

The caller can be running at any IRQL if *Address* points to nonpaged pool. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Available in Windows 2000 and later versions of Windows.

IRQL: Any level

RtlStoreUlonglong

Defined in: Wdm.h

The **RtlStoreUlonglong** macro stores a specified ULONGLONG value at a specified memory address, avoiding memory alignment faults.

Address [out]

PULONGLONG

A pointer to a location in which to store the specified ULONGLONG value.

Value [in]

ULONGLONG

The ULONGLONG value to be stored.

Return value

VOID

RtlStoreUlonglong avoids memory alignment faults. If the address specified by *Address* is not aligned to the storage requirements of a ULONGLONG, **RtlStoreUlonglong** stores the bytes of *Value* beginning at the memory location (PUCHAR)*Address*.

RtlStoreUlonglong runs at any IRQL if *Address* points to nonpaged pool; otherwise, it must run at IRQL <= APC_LEVEL.

Available starting with Windows 2000.

IRQL: Any level

RtlStoreUlongPtr

Defined in: Wdm.h

The **RtlStoreUlongPtr** macro stores a specified ULONG_PTR value at a specified memory location, avoiding memory alignment faults.

Address [out]

PULONG_PTR

A pointer to a location in which to store the ULONG_PTR value.

Value [in]

ULONG_PTR

Specifies the ULONG_PTR value to be stored.

Return value

VOID

RtlStoreUlongPtr avoids memory alignment faults. If the value of *Address* is not aligned to the storage requirements of a ULONG_PTR, **RtlStoreUlongPtr** stores the bytes of *Value* beginning at the memory location (PUCHAR)*Address*.

RtlStoreUlongPtr runs at any IRQL if *Address* points to nonpaged pool; otherwise it must run at IRQL <= APC_LEVEL.

Available in Windows 2000 and later versions of Windows.

IRQL: Any level

RtlStoreUshort

Defined in: Wdm.h

The **RtlStoreUshort** macro stores a USHORT value at a particular address, avoiding alignment faults.

Address [out]

PUSHORT

A pointer to a location in which to store the specified USHORT value.

Value [in]

USHORT

Specifies a USHORT value to be stored.

Return value

VOID

The caller can be running at any IRQL if *Address* points to nonpaged pool. Otherwise, the caller must be running at IRQL <= APC_LEVEL.

Available in Windows 2000 and later versions of Windows.

IRQL: Any level

WRITE_REGISTER_BUFFER_ULONG64

Defined in: Wdm.h

The **WRITE_REGISTER_BUFFER_ULONG64** macro writes a number of ULONG64 values from a buffer to the specified register.

Register [in]

PULONG64

Pointer to the register, which must be a mapped range in memory space.

Buffer [in]

PULONG64

Pointer to a buffer that an array of ULONG64 values is to be written to.

Count [in]

ULONG

Specifies the number of ULONG64 values to be written to the register.

Return value

VOID

The size of the *Buffer* buffer must be large enough to contain at least the specified number of ULONG64 values.

Callers of the **WRITE_REGISTER_BUFFER_ULONG64** macro can be running at any IRQL, assuming that the *Buffer* buffer is resident and the *Register* register is resident, mapped device memory.

Available only in 64-bit versions of Windows.

IRQL: Any level

WRITE_REGISTER_ULONG64

Defined in: Wdm.h

The **WRITE_REGISTER_ULONG64** macro writes a ULONG64 value to the specified address.

*volatile *Register [in]*

ULONG64

Pointer to the register, which must be a mapped range in memory space.

Value [in]

ULONG64

Specifies a ULONG64 value to write to the register.

Return value

VOID

Callers of the **WRITE_REGISTER_ULONG64** macro can be running at any IRQL, assuming the *Register* register is resident, mapped device memory.

Available only in 64-bit versions of Windows.

IRQL: Any level

ZwCurrentProcess

Defined in: Wdm.h

The **ZwCurrentProcess** macro returns a handle to the current process.

Return value

HANDLE

ZwCurrentProcess returns a special handle value that represents the current process.

The returned value is not a true handle, but it is a special value that always represents the current process.

NtCurrentProcess and **ZwCurrentProcess** are two versions of the same Windows Native System Services routine. The **NtCurrentProcess** routine in the Windows kernel is not directly accessible to kernel-mode drivers. However, kernel-mode drivers can access this routine indirectly by calling **ZwCurrentProcess**.

For calls from kernel-mode drivers, the **Nt_Xxx_** and **Zw_Xxx_** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt_Xxx_** and **Zw_Xxx_** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

All supported operating systems.

IRQL: Any level

ZwCurrentThread

Defined in: Wdm.h

The **ZwCurrentThread** macro returns a handle to the current thread.

Return value

HANDLE

ZwCurrentThread returns a special handle value that represents the current thread.

The returned value is not a true handle, but it is a special value that always represents the current thread.

NtCurrentThread and **ZwCurrentThread** are two versions of the same Windows Native System Services routine. The **NtCurrentThread** routine in the Windows kernel is not directly accessible to kernel-mode drivers. However, kernel-mode drivers can access this routine indirectly by calling the **ZwCurrentThread** routine.

For calls from kernel-mode drivers, the **Nt_Xxx_** and **Zw_Xxx_** versions of a Windows Native System Services routine can behave differently in the way that they handle and interpret input parameters. For more information about the relationship between the **Nt_Xxx_** and **Zw_Xxx_** versions of a routine, see Using Nt and Zw Versions of the Native System Services Routines.

All supported operating systems.

IRQL: Any level

Windows kernel opaque structures

6/25/2019 • 11 minutes to read • Edit Online

The following table contains Windows kernel opaque structures:

OPAQUE STRUCTURE	DESCRIPTION
EPROCESS	The **EPROCESS** structure is an opaque structure that serves as the process object for a process. Some routines, such as **PsGetProcessCreateTimeQuadPart**, use **EPROCESS** to identify the process to operate on. Drivers can use the **PsGetCurrentProcess** routine to obtain a pointer to the process object for the current process and can use the **ObReferenceObjectByHandle** routine to obtain a pointer to the process object that is associated with the specified handle. The **PsInitialSystemProcess** global variable points to the process object for the system process. Note that a process object is an Object Manager object. Drivers should use Object Manager routines such as **ObReferenceObject** and **ObDereferenceObject** to maintain the object's reference count. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
ETHREAD	The **ETHREAD** structure is an opaque structure that serves as the thread object for a thread. Some routines, such as **PsIsSystemThread**, use **ETHREAD** to identify the thread to operate on. Drivers can use the **PsGetCurrentThread** routine to obtain a pointer to the thread object for the current thread and can use the **ObReferenceObjectByHandle** routine to obtain a pointer to the thread object that is associated with the specified handle. Note that a thread object is an Object Manager object. Drivers should use Object Manager routines such as **ObReferenceObject** and **ObDereferenceObject** to maintain the object's reference count. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.

OPAQUE STRUCTURE	DESCRIPTION
EX_RUNDOWN_REF	The **EX_RUNDOWN_REF** structure is an opaque system structure that contains information about the status of run-down protection for an associated shared object. `typedef struct _EX_RUNDOWN_REF {` `  ...  // opaque` `} EX_RUNDOWN_REF, *PEX_RUNDOWN_REF;` The run-down protection routines all take a pointer to an **EX_RUNDOWN_REF** structure as their first parameter. These routines are listed at the bottom of this page. For more information, see Run-Down Protection. Header: Wdm.h. Include Wdm.h.
EX_TIMER	The **EX_TIMER** structure is an opaque structure that is used by the operating system to represent an **EX_TIMER** timer object. `typedef struct _EX_TIMER *PEX_TIMER;` All members of this structure are opaque to drivers. The following **Ex*Xxx*Timer** routines require a pointer to a system-allocated **EX_TIMER** structure as an input parameter: • **ExSetTimer** • **ExCancelTimer** • **ExDeleteTimer** **EX_TIMER**-based timer objects are created by the operating system. To get such a timer object, your driver calls the **ExAllocateTimer** routine. When this object is no longer needed, the driver is responsible for deleting the object by calling **ExDeleteTimer**. For more information, see Ex*Xxx*Timer Routines and EX_TIMER Objects. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
FAST_MUTEX	A **FAST_MUTEX** structure is an opaque data structure that represents a fast mutex. A **FAST_MUTEX** structure is initialized by the **ExInitializeFastMutex** routine. For more information about fast mutexes, see Fast Mutexes and Guarded Mutexes. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.

OPAQUE STRUCTURE	DESCRIPTION
IO_CSQ	The **IO_CSQ** structure is an opaque structure used to specify the driver's cancel-safe IRP queue routines. Do not set the members of this structure directly. Use **IoCsqInitialize** or **IoCsqInitializeEx** to initialize this structure. For an overview of how to use cancel-safe IRP queues, see Cancel-Safe IRP Queues. Available on Microsoft Windows XP and later versions of the Windows operating system. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
IO_CSQ_IRP_CONTEXT	The **IO_CSQ_IRP_CONTEXT** structure is an opaque data structure used to specify the IRP context for an IRP in the driver's cancel-safe IRP queue. It is used as a key by the **IoCsqInsertIrp**, **IoCsqInsertIrpEx**, and **IoCsqRemoveIrp** routines to identify particular IRPs in the queue. For an overview of how to use cancel-safe IRP queues, see Cancel-Safe IRP Queues. Available on Microsoft Windows XP and later versions of the Windows operating system. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
IO_WORKITEM	The **IO_WORKITEM** structure is an opaque structure that describes a work item for a system worker thread. A driver can allocate a work item by calling **IoAllocateWorkItem**. Alternatively, a driver can allocate its own buffer, and then call **IoInitializeWorkItem** to initialize that buffer as a work item. Any work item that is allocated by **IoAllocateWorkItem** must be freed by **IoFreeWorkItem**. Any memory that is initialized by **IoInitializeWorkItem** must be uninitialized by **IoUninitializeWorkItem** before it can be freed. For more information about work items, see System Worker Threads. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
KBUGCHECK_CALLBACK_RECORD	The **KBUGCHECK_CALLBACK_RECORD** structure is an opaque structure that is used by the **KeRegisterBugCheckCallback** and **KeDeregisterBugCheckCallback** routines. The **KBUGCHECK_CALLBACK_RECORD** structure is used for bookkeeping by the **KeRegisterBugCheckReasonCallback** and **KeDeregisterBugCheckReasonCallback** routines. The structure must be allocated in resident memory, such as nonpaged pool. Use the **KeInitializeCallbackRecord** routine to initialize the structure before using it. Header: Ntddk.h. Include: Ntddk.h.

OPAQUE STRUCTURE	DESCRIPTION
KBUGCHECK_REASON_CALLBACK_RECORD	The **KBUGCHECK_REASON_CALLBACK_RECORD** structure is an opaque structure that is used by the **KeRegisterBugCheckReasonCallback** and **KeDeregisterBugCheckReasonCallback** routines. The **KBUGCHECK_REASON_CALLBACK_RECORD** structure is used for bookkeeping by the **KeRegisterBugCheckReasonCallback** and **KeDeregisterBugCheckReasonCallback** routines. The structure must be allocated in resident memory, such as nonpaged pool. Use the **KeInitializeCallbackRecord** routine to initialize the structure before using it. Available on Microsoft Windows XP with Service Pack 1 (SP1), Windows Server 2003, and later versions of the Windows operating system. Header: Ntddk.h. Include: Ntddk.h.
KDPC	The **KDPC** structure is an opaque structure that represents a DPC object. Do not set the members of this structure directly. See DPC Objects and DPCs. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
KFLOATING_SAVE	The **KFLOATING_SAVE** structure is an opaque structure that describes the floating-point state that the **KeSaveFloatingPointState** routine saved. Use **KeRestoreFloatingPointState** to restore the floating-point state. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
KGUARDED_MUTEX	The **KGUARDED_MUTEX** structure is an opaque structure that represents a guarded mutex. Use **KeInitializeGuardedMutex** to initialize a **KGUARDED_MUTEX** structure as a guarded mutex. Guarded mutexes must be allocated from non-paged pool. For more information about guarded mutexes, see Fast Mutexes and Guarded Mutexes. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
KINTERRUPT	A **KINTERRUPT** structure is an opaque structure that represents an interrupt to the system. **IoConnectInterruptEx** provides a pointer to the **KINTERRUPT** structure for the interrupt when the driver registers an *InterruptService* or *InterruptMessageService* routine. The driver uses this pointer when acquiring or releasing the interrupt spin lock for the interrupt. The driver also uses this pointer when unregistering an *InterruptService* routine. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.

OPAQUE STRUCTURE	DESCRIPTION
KLOCK_QUEUE_HANDLE	The **KLOCK_QUEUE_HANDLE** structure is an opaque structure that describes a queued spin lock. The driver allocates the **KLOCK_QUEUE_HANDLE** structure, and passes it to **KeAcquireInStackQueuedSpinLock** and **KeAcquireInStackQueuedSpinLockAtDpcLevel** to acquire the queued spin lock. Those routines initialize the structure to represent the queued spin lock. The driver passes the structure to **KeReleaseInStackQueuedSpinLock** and **KeReleaseInStackQueuedSpinLockFromDpcLevel** when releasing the spin lock. For more information, see Queued Spin Locks. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
KTIMER	The **KTIMER** structure is an opaque structure that represents a timer object. Do not set the members of this structure directly. For more information, see Timer Objects and DPCs. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
LOOKASIDE_LIST_EX	The **LOOKASIDE_LIST_EX** structure describes a lookaside list. `typedef struct _LOOKASIDE_LIST_EX {` `  ...  // opaque` `} LOOKASIDE_LIST_EX, *PLOOKASIDE_LIST_EX;` A lookaside list is a pool of fixed-size buffers that the driver can manage locally to reduce the number of calls to system allocation routines and, thereby, to improve performance. The buffers are of uniform size and are stored as entries in the lookaside list. Drivers should treat the **LOOKASIDE_LIST_EX** structure as opaque. Drivers that access structure members or that have dependencies on the locations of these members might not remain portable and interoperable with other drivers. The following See Also section contains a list of the routines that use this structure. For more information about lookaside lists, see Using Lookaside Lists. On 64-bit platforms, this structure must be 16-byte aligned. Supported starting with Windows Vista. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.

OPAQUE STRUCTURE	DESCRIPTION
NPAGED_LOOKASIDE_LIST	The **NPAGED_LOOKASIDE_LIST** structure is an opaque structure that describes a lookaside list of fixed-size buffers allocated from nonpaged pool. The system creates new entries and destroys unused entries on the list as necessary. For fixed-size buffers, using a lookaside list is quicker than allocating memory directly. Use **ExInitializeNPagedLookasideList** to initialize the lookaside list. Use **ExAllocateFromNPagedLookasideList** to allocate a buffer from the list, and **ExFreeToNPagedLookasideList** to return a buffer to the list. Drivers must always explicitly free any lookaside lists they create before unloading. It is a serious programming error to do otherwise. Use **ExDeleteNPagedLookasideList** to free the list. Drivers can also use lookaside lists for paged pool. Starting with Windows 2000, a **PAGED_LOOKASIDE_LIST** structure describes a lookaside list that contains paged buffers. Starting with Windows Vista, a **LOOKASIDE_LIST_EX** structure can describe a lookaside list that contains either paged or nonpaged buffers. For more information, see Using Lookaside Lists. On 64-bit platforms, this structure must be 16-byte aligned. Supported starting with Windows 2000. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
OBJECT_TYPE	**OBJECT_TYPE** is an opaque structure that specifies the object type of a handle. For more information, see **ObReferenceObjectByHandle**. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.

OPAQUE STRUCTURE	DESCRIPTION
PAGED_LOOKASIDE_LIST	The **PAGED_LOOKASIDE_LIST** structure is an opaque structure that describes a lookaside list of fixed-size buffers allocated from paged pool. The system creates new entries and destroys unused entries on the list as necessary. For fixed-size buffers, using a lookaside list is quicker than allocating memory directly. Use **ExInitializePagedLookasideList** to initialize the lookaside list. Use **ExAllocateFromPagedLookasideList** to allocate a buffer from the list, and **ExFreeToPagedLookasideList** to return a buffer to the list. Drivers must always explicitly free any lookaside lists they create before unloading. It is a serious programming error to do otherwise. Use **ExDeletePagedLookasideList** to free the list. Drivers can also use lookaside lists for nonpaged pool. Starting with Windows 2000, an **NPAGED_LOOKASIDE_LIST** structure describes a lookaside list that contains nonpaged buffers. Starting with Windows Vista, a **LOOKASIDE_LIST_EX** structure can describe a lookaside list that contains either paged or nonpaged buffers. For more information, see Using Lookaside Lists. On 64-bit platforms, this structure must be 16-byte aligned. Supported starting with Windows 2000. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.

OPAQUE STRUCTURE	DESCRIPTION
RTL_BITMAP	The **RTL_BITMAP** structure is an opaque structure that describes a bitmap. ``` typedef struct _RTL_BITMAP { // opaque } RTL_BITMAP, *PRTL_BITMAP; ``` Do not directly access the members of this structure. Drivers that have dependencies on member locations or that access member values directly might not remain compatible with future versions of the Windows operating system. The **RTL_BITMAP** structure serves as a header for a general-purpose, one-dimensional bitmap of arbitrary length. A driver can use such a bitmap as an economical way to keep track of a set of reusable items. For example, a file system can use bitmaps to track which clusters and sectors on a hard disk have already been allocated to hold file data. For a list of the **Rtl*Xxx*** routines that use **RTL_BITMAP** structures, see the following See Also section. The caller of these **Rtl*Xxx*** routines is responsible for allocating the storage for the **RTL_BITMAP** structure and for the buffer that contains the bitmap. This buffer must begin on a four-byte boundary in memory and must be a multiple of four bytes in length. The bitmap begins at the start of the buffer but can contain any number of bits that will fit in the allocated buffer. Before supplying an **RTL_BITMAP** structure as a parameter to an **Rtl*Xxx*** routine, call the **RtlInitializeBitMap** routine to initialize the structure. The input parameters to this routine are a pointer to a buffer that contains the bitmap, and the size, in bits, of the bitmap. **RtlInitializeBitMap** does not change the contents of this buffer. If the caller allocates the storage for the **RTL_BITMAP** structure and bitmap in paged memory, the caller must be running at IRQL <= APC_LEVEL when it passes a pointer to this structure as a parameter to any of the **Rtl*Xxx*** routines that are listed in the See Also section. If the caller allocates the storage from nonpaged memory (or, equivalently, from paged memory that is locked), the caller can be running at any IRQL when it calls the **Rtl*Xxx*** routine. Supported in Windows 2000 and later versions of Windows. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.

OPAQUE STRUCTURE	DESCRIPTION
RTL_RUN_ONCE	The **RTL_RUN_ONCE** structure is an opaque structure that stores the information for a one-time initialization. Drivers must initialize this structure by calling the **RtlRunOnceInitialize** routine before passing it to any other **RtlRunOnce*Xxx*** routines. Available only on Windows Vista and later versions of the Windows operating system. Header: Ntddk.h. Include: Ntddk.h.
SECURITY_SUBJECT_CONTEXT	The **SECURITY_SUBJECT_CONTEXT** structure is an opaque structure that represents the security context within which a particular operation is taking place. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
SLIST_HEADER	An **SLIST_HEADER** structure is an opaque structure that serves as the header for a sequenced singly linked list. For more information, see Singly and Doubly Linked Lists. On 64-bit platforms, **SLIST_HEADER** structures must be 16-byte aligned. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.
XSTATE_SAVE	The **XSTATE_SAVE** structure is an opaque structure that describes the extended processor state information that a kernel-mode driver saves and restores. `typedef struct _XSTATE_SAVE {` `  ...  // opaque` `} XSTATE_SAVE, *PXSTATE_SAVE;` All members are opaque. This structure is used by the **KeSaveExtendedProcessorState** and **KeRestoreExtendedProcessorState** routines. Supported in Windows 7 and later versions of the Windows operating system. Header: Wdm.h. Include: Wdm.h, Ntddk.h, Ntifs.h.

Related topics

BugCheckDumpIoCallback
BugCheckSecondaryDumpDataCallback
ExAcquireFastMutex
ExAcquireFastMutexUnsafe
ExAllocateFromLookasideListEx
ExAllocateFromNPagedLookasideList
ExAllocateFromPagedLookasideList
ExAllocateTimer
ExDeletePagedLookasideList

ExFreeToPagedLookasideList
ExInitializePagedLookasideList
ExCancelTimer
ExDeleteLookasideListEx
ExDeleteNPagedLookasideList
ExDeleteTimer
ExFlushLookasideListEx
ExFreeToLookasideListEx
ExFreeToNPagedLookasideList
ExInitializeLookasideListEx
ExInitializeNPagedLookasideList
ExInitializeSListHead
ExInterlockedFlushSList
ExInterlockedPopEntrySList
ExInterlockedPushEntrySList
ExQueryDepthSList
ExReleaseFastMutex
ExReleaseFastMutexUnsafe
ExSetTimer
ExTryToAcquireFastMutex
ExTimerCallback
IoAllocateWorkItem
IoConnectInterruptEx
IoCsqInitialize
IoCsqInitializeEx
IoCsqInsertIrp
IoCsqInsertIrpEx
IoCsqRemoveIrp
IoDisconnectInterruptEx
IoFreeWorkItem
IoInitializeWorkItem
IoRequestDpc
IoUninitializeWorkItem
KeAcquireGuardedMutex
KeAcquireGuardedMutexUnsafe
KeAcquireInStackQueuedSpinLock
KeAcquireInStackQueuedSpinLockAtDpcLevel
KeAcquireInterruptSpinLock
KeCancelTimer
KeInitializeCallbackRecord
KeInitializeGuardedMutex
KeInitializeTimer
KeInitializeTimerEx
KeReadStateTimer
KeRestoreExtendedProcessorState
KeSaveExtendedProcessorState
KeSetTimer
KeSetTimerEx
KeDeregisterBugCheckCallback
KeDeregisterBugCheckReasonCallback
KeInsertQueueDpc
KeRegisterBugCheckCallback

KeRegisterBugCheckReasonCallback
KeReleaseGuardedMutexUnsafe
KeReleaseInStackQueuedSpinLock
KeReleaseInStackQueuedSpinLockFromDpcLevel
KeReleaseInterruptSpinLock
KeRestoreFloatingPointState
KeSaveFloatingPointState
KeSynchronizeExecution
LookasideListAllocateEx
LookasideListFreeEx
ObReferenceObjectByHandle
PsGetCurrentProcess
PsGetProcessCreateTimeQuadPart
PsInitialSystemProcess
PsIsSystemThread
RtlRunOnceBeginInitialize
RtlRunOnceComplete
RtlRunOnceExecuteOnce
RtlRunOnceInitialize
RunOnceInitialization
Run-Down Protection
SeAccessCheck
SeAssignSecurity
SeAssignSecurityEx

Working with the GUID_DEVICE_RESET_INTERFACE_STANDARD

12/5/2018 • 6 minutes to read • Edit Online

The GUID_DEVICE_RESET_INTERFACE_STANDARD interface defines a standard way for function drivers to attempt to reset and recover a malfunctioning device.

Two types of device resets are available through this interface:

- Function-level device reset. In this case, the reset operation is restricted to a specific device, and is not visible to other devices. The device stays connected to the bus throughout the reset and returns to a valid state (initial state) after the reset. This type of reset has the least impact on the system.
- This type of reset can be implemented either by the bus driver or by ACPI firmware. The bus driver can implement a function-level reset if the bus specification defines an in-band reset mechanism that meets the requirement. ACPI firmware can optionally override a bus driver-defined function-level reset with its own implementation.

Platform-level device reset. In this case, the reset operation causes the device to be reported as missing from the bus. The reset operation affects a specific device and all other devices that are connected to it via the same power rail or reset line. This type of reset has the most impact on the system. The OS will tear down and rebuild the stacks of all affected devices to ensure that everything restarts from a blank state.

Starting in Windows 10, these registry entries under the `HKLM\SYSTEM\CurrentControlSet\Control\Pnp` key configures the reset operation:

- DeviceResetRetryInterval: Time period before the reset operation starts. Default value is 3 seconds. Minimum value is 100 milliseconds; maximum value is 30 seconds.
- DeviceResetMaximumRetries: Number of times the reset operation is attempted.

> **NOTE**
> The GUID_DEVICE_RESET_INTERFACE_STANDARD interface is available starting in Windows 10.

Using the device reset interface

If a function driver detects that the device is not functioning correctly, it should first attempt a function-level reset. If a function-level reset does not fix the issue, then the driver may choose to attempt a platform-level reset. However, a platform-level reset should only be used as the final option.

To query for this interface, a device driver sends an IRP_MN_QUERY_INTERFACE IRP down the driver stack. For this IRP, the driver sets the InterfaceType input parameter to GUID_DEVICE_RESET_INTERFACE_STANDARD. On successful completion of the IRP, the Interface output parameter is a pointer to a DEVICE_RESET_INTERFACE_STANDARD structure. This structure contains a pointer to the DeviceReset routine, which can be used to request a function-level or platform-level reset.

Supporting the device reset interface in function drivers

To support the device reset interface, the device stack must meet the following requirements.

The function driver must properly handle IRP_MN_QUERY_REMOVE_DEVICE, IRP_MN_REMOVE_DEVICE and IRP_MN_SURPRISE_REMOVAL.

In most cases, when the driver receives IRP_MN_QUERY_REMOVE_DEVICE, it should return a success so that the device can be safely removed. However, there may be cases where the device cannot be safely stopped, such as if the device is stuck in a loop writing to a memory buffer. In such cases, the driver should return STATUS_DEVICE_HUNG to IRP_MN_QUERY_REMOVE_DEVICE. The PnP manager will continue the IRP_MN_QUERY_REMOVE_DEVICE and IRP_MN_REMOVE_DEVICE process, but that particular stack will not receive IRP_MN_REMOVE_DEVICE. Instead, the device stack will receive IRP_MN_SURPRISE_REMOVAL after the device has been reset.

For more information about these IRPs, see:

Handling an IRP_MN_QUERY_REMOVE_DEVICE Request

Handling an IRP_MN_REMOVE_DEVICE Request

Handling an IRP_MN_SURPRISE_REMOVAL Request

Supporting the device reset interface in filter drivers

Filter drivers may intercept IRP_MN_QUERY_INTERFACE IRPs that have the GUID_DEVICE_RESET_INTERFACE_STANDARD interface type. By doing so, they can continue to delegate to the GUID_DEVICE_RESET_INTERFACE_STANDARD interface but perform device-specific operations before or after the reset operation. Alternatively, they can override the GUID_DEVICE_RESET_INTERFACE_STANDARD interface returned by the bus driver with its own interface in order to provide its own reset operation.

Supporting the device reset interface in bus drivers

Bus drivers that participate in the device reset process (that is, bus drivers that are associated with the device that is requesting the reset and bus drivers that are associated with devices that are responding to the reset request) must meet one of the following requirements:

- Be hot plug capable. The bus driver must be able to detect a device being removed from the bus without notice, and a device being plugged into the bus.
- Alternatively, it must implement the GUID_REENUMERATE_SELF_INTERFACE_STANDARD interface. This simulates a device being pulled from a bus and being plugged back in. Built-in bus drivers (such as PCI and SDBUS) support this interface. Therefore, if the device being reset uses one of these buses, no bus driver modifications should be necessary.

For WDF-based bus drivers, the WDF framework registers the GUID_REENUMERATE_SELF_INTERFACE_STANDARD interface on behalf of the drivers. Therefore, registering this interface is not necessary for those drivers. If the bus driver needs to do perform some operations before its child devices are re-enumerated, it must register for the EvtChildListDeviceReenumerated callback routine and perform the operations in that routine. Because this callback routine may be called in parallel for all PDO's, the code in the routine may need to protect against race conditions.

ACPI firmware: Function-level reset

To support function-level device reset, there must be an _RST method defined inside the Device scope. If present, this method will override the bus driver's implementation of function-level device reset (if present) for that device. When executed, the _RST method must reset only that device, and must not affect other devices. In addition, the device must stay connected on the bus.

ACPI firmware: Platform-level reset

To support platform-level device reset, there are two options:

- The ACPI firmware can define a PowerResource that implements the _RST method, and all devices that are affected by this reset method can refer to this PowerResource through a _PRR object defined under their Device scope.
- The device can declare a _PR3 object. In this case, the ACPI driver will use D3cold power cycling to perform the reset, and reset dependencies between devices will be determined from the _PR3 object.

If the _PRR object exists in the Device scope, the ACPI driver will use the _RST method in the referenced PowerResource to perform the reset. If no _PRR object is defined but the _PR3 object is defined, then the ACPI driver will use D3cold power cycling to perform the reset. If neither the _PRR or _PR3 object is defined, then the device does not support a platform-level reset and the ACPI driver will report that the platform-level reset is not available.

Verifying ACPI firmware on the test system

To test your driver that supports device reset and recovery, follow this procedure. This procedure assumes you are using this example ASL file.

```
DefinitionBlock("SSDT.AML", "SSDT", 0x01, "XyzOEM", "TestTabl", 0x00001000)
{
   Scope(\_SB_)
      {
       PowerResource(PWFR, 0x5, 0x0)
       {
           Method(_RST, 0x0, NotSerialized)    { }

           // Placeholder methods as power resources need _ON, _OFF, _STA.
           Method(_STA, 0x0, NotSerialized)
           {
               Return(0xF)
           }

           Method(_ON_, 0x0, NotSerialized)    { }

           Method(_OFF, 0x0, NotSerialized)    { }

       } // PowerResource()
   } // Scope (\_SB_)

   // Assumes WiFi device is declared under \_SB.XYZ.
   Scope(\_SB_.XYZ.WIFI)
       {

       // Declare PWFR as WiFi reset power rail
       Name(_PRR, Package(One)
           {
               \_SB_.PWFR
           })
       } // Scope (\_SB)
}
```

1. Compile the test ASL file to an AML by using an ASL compiler, such as Asl.exe. The executable in included in the Windows Driver Kit (WDK).

```
Asl <test>.asl
```

The preceding command generates SSDT.aml.

2. Rename SSDT.aml to acpitabl.dat.
3. Copy acpitabl.dat to %systemroot%\system32 on the test system.
4. Enable test signing on the test system.

```
bcdedit /set GUID_DEVICE_RESET_INTERFACE_STANDARD testsigning on
```

5. Reboot the test system.
6. Verify that the table is loaded. In Windows Debugger, use these commands.

- !acpicache
- dt _DESCRIPTION_HEADER address of the SSDT table

```
0: kd> !acpicache
Dumping cached ACPI tables...
  SSDT @(ffffffffffd03018) Rev: 0x1 Len: 0x000043 TableID: TestTabl
  XSDT @(ffffffffffd05018) Rev: 0x1 Len: 0x000114 TableID: HSW-FFRD
       ...
       ...

0: kd> dt _DESCRIPTION_HEADER ffffffffffd03018
ACPI!_DESCRIPTION_HEADER
   +0x000 Signature        : 0x54445353
   +0x004 Length           : 0x43
   +0x008 Revision         : 0x1 ''
   +0x009 Checksum         : 0x37 '7'
   +0x00a OEMID            : [6]  "XyzOEM"
   +0x010 OEMTableID       : [8]  "TestTabl"
   +0x018 OEMRevision      : 0x1000
   +0x01c CreatorID        : [4]  "MSFT"
   +0x020 CreatorRev       : 0x5000000
```

See Also

_DEVICE_RESET_INTERFACE_STANDARD