MICROSOFT® PROFESSIONAL EDITIONS

Microsoft®

The comprehensive, must-have reference for anyone who develops drivers for Windows 2000

Microsoft® Windows® 2000 Driver Development Reference

Volume 2

Microsoft®

Driver Development Reference

Volume 2

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data pending.

Printed and bound in the United States of America.

2 3 4 5 6 7 8 9 WCWC 5 4 3 2 1 0

Distributed in Canada by Penguin Books Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com.

Acquisitions Editor: Ben Ryan
Project Management and Production: Online Training Solutions, Inc.
Project Editor: John Pierce

Acknowledgments to: the Microsoft Corporation Windows 2000 Team

Part No. 097-0002734

Contents

PART 1

Kernel-Mode Support Routines

CHAPTER 1

Summary of Kernel-Mode Support Routines

This chapter summarizes the kernel-mode support routines that can be called by Microsoft® Windows NT®/Windows® 2000 and WDM kernel-mode drivers. Drivers can also use routines provided by a compiler, such as C string manipulation routines.

Support routines are categorized as follows:

- Initialization and unload
- IRPs
- Synchronization
- Memory
- DMA
- PIO
- Driver-managed queues
- Driver-dedicated system threads and system worker threads
- Strings
- Data conversions
- Access to and access rights on driver-managed objects
- Handling errors

Some routines are listed in more than one section or subsection of this chapter.

Initialization and Unload

This section summarizes kernel-mode support routines that can be called by drivers from their **DriverEntry**, **AddDevice**, **Reinitialize**, or **Unload** routines.

The categories of kernel-mode support routines include those that drivers can call to:

- Get and report hardware configuration information about their devices and the current platform
- Get and report configuration information and register interfaces in the registry
- Set up certain standard driver routines
- Set up and free the objects and resources they might use
- Initialize driver-managed internal queues

Hardware Configuration

IoGetDeviceProperty

Retrieves device setup information from the registry. Use this routine, rather than accessing the registry directly, to insulate a driver from differences across platforms and from possible changes in the registry structure.

IoReportDetectedDevice

Reports a nonPnP device to the PnP Manager.

IoReportResourceForDetection

Claims hardware resources in the configuration registry for a legacy device. This routine is for drivers that detect legacy hardware which cannot be enumerated by PnP.

IoGetDmaAdapter

Returns a pointer to the DMA adapter structure that represents either the DMA channel to which a device is connected or the driver's busmaster adapter.

IoGetConfigurationInformation

Returns a pointer to the I/O Manager's configuration information structure, which indicates the number of disk, floppy, CD-ROM, tape, SCSI HBAs, serial, and parallel device objects that have already been named by previously loaded drivers, as well as whether certain address ranges have been claimed by "AT" disk-type drivers.

HalExamineMBR

Returns data from the master boot record (MBR) of a disk.

IoReadPartitionTable

Returns a list of partitions on a disk with a given sector size.

IoInvalidateDeviceRelations

Notifies the PnP Manager that the relations for a device have changed. The types of device relations include bus relations, ejection relations, removal relations, and the target device relation.

IoInvalidateDeviceState

Notifies the PnP Manager that some aspect of the PnP state of a device has changed. In response, the PnP Manager sends an IRP_MN_QUERY_PNP_DEVICE_STATE to the device stack.

IoRegisterPlugPlayNotification

Registers a driver callback routine to be called when a PnP event of the specified category occurs.

IoUnregisterPlugPlayNotification

Removes the registration of a driver's callback routine for a PnP event.

IoRequestDeviceEject

Notifies the PnP Manager that the device eject button was pressed. This routine reports a request for device eject, not media eject.

IoReportTargetDeviceChange

Notifies the PnP Manager that a custom event has occurred on a device. The PnP Manager sends notification of the event to drivers that registered for notification on the device.

Registry

IoGetDeviceProperty

Retrieves device setup information from the registry. Use this routine, rather than accessing the registry directly, to insulate a driver from differences across platforms and from possible changes in the registry structure.

IoOpenDeviceInterfaceRegistryKey

Returns a handle to a registry key for storing information about a particular device interface.

IoOpenDeviceRegistryKey

Returns a handle to a device-specific or a driver-specific registry key for a particular device instance.

IoRegisterDeviceInterface

Registers device functionality (a device interface) that a driver will enable for use by applications or other system components. The I/O Manager creates a registry key for the device interface. Drivers can access persistent storage under this key using **IoOpenDeviceInterfaceRegistryKey**.

IoSetDeviceInterfaceState

Enables or disables a previously registered device interface. Applications and other system components can open only interfaces that are enabled.

RtlCheckRegistryKey

Returns STATUS_SUCCESS if a key exists in the registry along the given relative path.

RtlCreateRegistryKey

Adds a key object in the registry along the given relative path.

RtlQueryRegistryValues

Gives the driver-supplied QueryRegistry callback (read only) access to the entries for the specified value name along the specified relative path in the registry after the QueryRegistry routine is given control.

RtlWriteRegistryValue

Writes caller-supplied data into the registry along the specified relative path at the given value name.

RtlDeleteRegistryValue

Removes the specified value name (and the associated value entries) from the registry along the given relative path.

InitializeObjectAttributes

Sets up a parameter of type OBJECT_ATTRIBUTES for a subsequent call to a **ZwCreate***Xxx* or **ZwOpen***Xxx* routine.

ZwCreateKey

Creates a new key in the registry with the given object's attributes, allowed access, and creation options (such as whether the key is created again when the system is booted). Alternatively, opens an existing key and returns a handle for the key object.

ZwOpenKey

Returns a handle for a key in the registry given the object's attributes (which must include a name for the key) and the desired access to the object.

ZwQueryKey

Returns information about the class of a key, and the number and sizes of its subkeys. This information includes, for example, the length of subkey names and the size of value entries.

ZwEnumerateKey

Returns the specified information about the subkeys of an opened key in the registry.

ZwEnumerateValueKey

Returns the specified information about the value entry, as selected by a zero-based index, of an opened key in the registry.

ZwQueryValueKey

Returns the value entry, as selected by a zero-based index, for an opened key in the registry.

ZwSetValueKey

Replaces (or creates) a value entry for an opened key in the registry.

ZwFlushKey

Forces changes made by **ZwCreateKey** or **ZwSetValueKey** for the opened key object to be written to disk.

ZwDeleteKey

Removes a key and its value entries from the registry as soon as the key is closed.

ZwClose

Releases the handle for an opened object, causing the handle to become invalid and decrementing the reference count of the object handle.

Standard Driver Routines

IoRegisterDriverReinitialization

Sets up the driver-supplied Reinitialize routine, together with its context, so that the Reinitialize routine is called after each subsequently loaded driver's **DriverEntry** routine returns control.

IoConnectInterrupt

Registers an ISR and sets up interrupt objects using values supplied in the PnP IRP_MN_START_DEVICE request. Returns a pointer to a set of interrupt objects that must be passed, along with the driver's SynchCritSection entry point, to **KeSynchronizeExecution**.

IoDisconnectInterrupt

Releases a driver's interrupt objects.

IoInitializeDpcRequest

Associates a driver-supplied DpcForIsr routine with a given device object, so that the DpcForIsr can complete interrupt-driven I/O operations.

KeInitializeDpc

Initializes a DPC object, setting up a driver-supplied CustomDpc routine that can be called with a given context.

KeInitializeTimer

Initializes a notification timer object to the Not-Signaled state.

KeInitializeTimerEx

Initializes a notification or synchronization timer object to the Not-Signaled state.

IoInitializeTimer

Associates a timer with the given device object and registers a driver-supplied IoTimer routine for the device object.

MmLockPagableCodeSection

Locks a set of driver routines marked with a special compiler directive into system space. This operation can occur during driver initialization but usually occurs in the driver's DispatchCreate routine.

MmLockPagableDataSection

Locks a named data section, which is marked with a special compiler directive, into system space if that data is used infrequently, predictably, and at an IRQL less than DISPATCH_LEVEL.

MmLockPagableSectionByHandle

Locks a pageable section into system memory using a handle returned from **MmLockPagableCodeSection** or **MmLockPagableDataSection**.

MmUnlockPagableImageSection

Releases a set of driver routines or a set of data that was locked into nonpaged system space when the driver is no longer processing IRPs.

MmPageEntireDriver

Allows a driver to page out all of its code and data, regardless of the attributes of the various sections in the driver's image.

MmResetDriverPaging

Resets a driver's pageable status to that specified by the sections which make up the driver's image.

Objects and Resources

IoCreateDevice

Initializes a device object, which represents a physical, virtual, or logical device for which the driver is being loaded into the system. Then it allocates space for the driver-defined device extension associated with the device object.

IoDeleteDevice

Removes a device object from the system when the underlying device is removed from the system.

IoGetDeviceObjectPointer

Requests access to a named device object and returns a pointer that device object if the requested access is granted. Also returns a pointer to the file object referenced by the named device object. In effect, this routine establishes a connection between the caller and the next-lower-level driver.

IoAttachDeviceToDeviceStack

Attaches the caller's device object to the highest device object in a chain of drivers and returns a pointer to the previously highest device object. I/O requests bound for the target device are routed first to the caller.

IoGetAttachedDeviceReference

Returns a pointer to the highest level device object in a driver stack and increments the reference count on that object.

IoDetachDevice

Releases an attachment between the caller's device object and a target driver's device object.

IoAllocateDriverObjectExtension

Allocates a per-driver context area with a given unique identifier.

IoGetDriverObjectExtension

Retrieves a previously allocated per-driver context area.

IoRegisterDeviceInterface

Registers device functionality (a device interface) that a driver will enable for use by applications or other system components. The I/O Manager creates a registry key for the device interface. Drivers can access persistent storage under this key using **IoOpenDeviceInterfaceRegistryKey**.

IoIsWdmVersionAvailable

Checks whether a given WDM version is supported by the operating system.

IoDeleteSymbolicLink

Releases a symbolic link between a device object name and a user-visible name.

IoAssignArcName

Sets up a symbolic link between a named device object (such as a tape, floppy, or CD-ROM) and the corresponding ARC name for the device.

IoDeassignArcName

Releases the symbolic link created by calling **IoAssignArcName**.

IoSetShareAccess

Sets the access allowed to a given file object that represents a device. (Only highest-level drivers can call this routine.)

IoConnectInterrupt

Registers a driver's ISR according to the parameters supplied in the IRP_MN_START_ DEVICE request. Returns a pointer to a set of allocated, initialized, and connected interrupt objects that is used as an argument to **KeSynchronizeExecution**.

IoDisconnectInterrupt

Releases a driver's interrupt objects when the driver unloads.

IoReadPartitionTable

Returns a list of partitions on a disk with a given sector size.

IoSetPartitionInformation

Sets the partition type and number for a (disk) partition.

IoWritePartitionTable

Writes partition tables for a disk, given the device object that represents the disk, the sector size, and a pointer to a buffer containing the drive layout structure.

IoCreateController

Initializes a controller object that represents a physical device controller which is shared by two or more similar devices that have the same driver, and specifies the size of the controller extension.

IoDeleteController

Removes a controller object from the system.

KeInitializeSpinLock

Initializes a variable of type KSPIN_LOCK.

KeInitializeDpc

Initializes a DPC object, setting up a driver-supplied CustomDpc routine that can be called with a given context.

KeInitializeTimer

Initializes a notification timer object to the Not-Signaled state.

KeInitializeTimerEx

Initializes a notification or synchronization timer object to the Not-Signaled state.

KeInitializeEvent

Initializes an event object as a synchronization (single waiter) or notification (multiple waiters) type event and sets up its initial state (Signaled or Not-Signaled).

ExInitializeFastMutex

Initializes a fast mutex variable that is used to synchronize mutually exclusive access to a shared resource by a set of threads.

KeInitializeMutex

Initializes a mutex object at a given level number as set to the Signaled state.

KeInitializeSemaphore

Initializes a semaphore object to a given count and specifies an upper bound for the count.

IoCreateNotificationEvent

Initializes a named notification event to be used to synchronize access between two or more components. Notification events are not automatically reset.

IoCreateSynchronizationEvent

Initializes a named synchronization event to be used to serialize access to hardware between two otherwise unrelated drivers.

PsCreateSystemThread

Creates a kernel-mode thread that is associated with a given process object or with the default system process. Returns a handle for the thread.

PsTerminateSystemThread

Terminates the current thread and satisfies as many waits as possible for the current thread object.

KeSetBasePriorityThread

Sets up the run-time priority, relative to the system process, for a driver-created thread.

KeSetPriorityThread

Sets up the run-time priority for a driver-created thread with a real-time priority attribute.

MmIsThisAnNtAsSystem

Returns TRUE if the current platform is a server, indicating that more resources are likely to be necessary to process I/O requests than if the machine were a client.

MmQuerySystemSize

Returns an estimate (small, medium, or large) of the amount of memory available on the current platform.

ExInitializeNPagedLookasideList

Initializes a lookaside list of nonpaged memory. After a successful initialization, fixed-size blocks can be allocated from and freed to the lookaside list.

ExInitializePagedLookasideList

Initializes a lookaside list of paged memory. After a successful initialization, fixed-size blocks can be allocated from and freed to the lookaside list.

ExInitializeResourceLite

Initializes a resource, for which the caller provides the storage, to be used for synchronization by a set of threads.

ExReinitializeResourceLite

Reinitializes an existing resource variable.

ExDeleteResourceLite

Deletes a caller-initialized resource from the system's resource list.

ObReferenceObjectByHandle

Returns a pointer to the object body and handle information (attributes and granted access rights), given the handle for an object, the object's type, and a mask. Specifies the desired access to the object and the preferred access mode. A successful call increments the reference count for the object.

ObReferenceObjectByPointer

Increments the reference count for an object so the caller can ensure that the object is not removed from the system while the caller is using it.

ObReferenceObject

Increments the reference count for an object, given a pointer to the object.

ObDereferenceObject

Releases a reference to an object (decrements the reference count), given a pointer to the object body.

RtlInitString

Initializes a counted string in a buffer.

RtlInitAnsiString

Initializes a counted ANSI string in a buffer.

RtlInitUnicodeString

Initializes a counted Unicode string in a buffer.

InitializeObjectAttributes

Initializes a parameter of type OBJECT_ATTRIBUTES for a subsequent call to a **ZwCreate***Xxx* or **ZwOpen***Xxx* routine.

ZwCreateDirectoryObject

Creates or opens a directory object with a specified set of object attributes and requests one or more types of access for the caller. Returns a handle for the directory object.

ZwCreateFile

Creates or opens a file object that represents a physical, logical, or virtual device, a directory, a data file, or a volume. Returns a handle for the file object.

ZwCreateKey

Creates or opens a key object in the registry and returns a handle for the key object.

ZwDeleteKey

Deletes an existing, open key in the registry after the last handle for the key is closed.

ZwMakeTemporaryObject

Resets the "permanent" attribute of an opened object, so that the object and its name can be deleted when the reference count for the object becomes zero.

ZwClose

Releases the handle for an opened object, causing the handle to become invalid, and decrements the reference count of the object handle.

PsGetVersion

Indicates whether the driver is running on a free or checked build of Windows NT/ Windows 2000, and optionally supplies information about the operating system version and build number.

ObGetObjectSecurity

Returns a buffered security descriptor for a given object.

ObReleaseObjectSecurity

Releases the security descriptor returned by **ObGetObjectSecurity**.

Initializing Driver-Managed Queues

KeInitializeSpinLock

Initializes a variable of type KSPIN_LOCK. An initialized spin lock is a required parameter to the **Ex..InterlockedList** routines.

InitializeListHead

Sets up a queue header for a driver's internal queue, given a pointer to driver-supplied storage for the queue header and queue.

ExInitializeSListHead

Sets up the queue header for a sequenced, interlocked, singly-linked list.

KeInitializeDeviceQueue

Initializes a device queue object to a Not Busy state, setting up an associated spin lock for multiprocessor-safe access to device queue entries.

IRPs

This section describes kernel-mode support routines that drivers can call:

- While processing IRPs
- To allocate and set up IRPs for requests from higher-level drivers to lower drivers
- To use file objects

Processing IRPs

IoGetCurrentIrpStackLocation

Returns a pointer to the caller's I/O stack location in a given IRP.

IoGetNextIrpStackLocation

Returns a pointer to the next-lower-level driver's I/O stack location in a given IRP.

IoCopyCurrentIrpStackLocationToNext

Copies the IRP stack parameters from the current stack location to the stack location of the next-lower driver and allows the current driver to set an I/O completion routine.

IoSkipCurrentIrpStackLocation

Copies the IRP stack parameters from the current stack location to the stack location of the next-lower driver and does not allow the current driver to set an I/O completion routine.

IoGetRelatedDeviceObject

Returns a pointer to the device object represented by a given file object.

IoGetFunctionCodeFromCtlCode

Returns the value of the function field within a given IOCTL_*XXX* or FSCTL_*XXX*.

IoSetCompletionRoutine

Registers a driver-supplied IoCompletion routine for an IRP, so the IoCompletion routine is called when the next-lower-level driver has completed the requested operation in one or more of the following ways: successfully, with an error, or by canceling the IRP.

IoCallDriver

Sends an IRP to a lower-level driver.

PoCallDriver

Sends an IRP with major function code IRP_MJ_POWER to the next-lower driver.

IoMarkIrpPending

Marks a given IRP indicating that STATUS_PENDING was returned because further processing is required by another driver routine or by a lower-level driver.

IoStartPacket

Calls the driver's StartIo routine with the given IRP for the given device object or inserts the IRP into the device queue if the device is already busy, specifying whether the IRP is cancelable.

IoAcquireCancelSpinLock

Synchronizes cancelable state transitions for IRPs in a multiprocessor-safe manner.

IoSetCancelRoutine

Sets or clears the Cancel routine in an IRP. Setting a Cancel routine makes an IRP cancelable.

IoReleaseCancelSpinLock

Releases the cancel spin lock when the driver has changed the cancelable state of an IRP or releases the cancel spin lock from the driver's Cancel routine.

IoCancelIrp

Marks an IRP as canceled.

IoReadPartitionTable

Returns a list of partitions on a disk with a given sector size.

IoSetPartitionInformation

Sets the partition type and number for a (disk) partition.

IoWritePartitionTable

Writes partition tables for a disk, given the device object representing the disk, the sector size, and a pointer to a buffer containing the drive geometry.

IoAllocateErrorLogEntry

Allocates and initializes an error log packet; returns a pointer so that the caller can supply error-log data and call **IoWriteErrorLogEntry** with the packet.

IoWriteErrorLogEntry

Queues a previously allocated and filled-in error log packet to the system error logging thread.

IoIsErrorUserInduced

Returns a Boolean value indicating whether an I/O request failed due to one of the following conditions: STATUS_IO_TIMEOUT, STATUS_DEVICE_NOT_READY, STATUS_UNRECOGNIZED_MEDIA, STATUS_VERIFY_REQUIRED, STATUS_WRONG_VOLUME, STATUS_MEDIA_WRITE_PROTECTED, or STATUS_NO_MEDIA_IN_DEVICE. If the result is TRUE, a removable-media driver must call **IoSetHardErrorOrVerifyDevice** before completing the IRP.

IoSetHardErrorOrVerifyDevice

Supplies the device object for which the given IRP was failed due to a user-induced error, such as supplying the incorrect media for the requested operation or changing the media before the requested operation was completed. A file system driver uses the associated device object to notify the user, who can then correct the error or retry the operation.

IoGetDeviceToVerify

Returns a pointer to the device object, representing a removable-media device that is the target of the given thread's I/O request. Useful only to file systems or other highest-level drivers.

IoRaiseHardError

Notifies the user that the given IRP was failed on the given device object for an optional VPB, so that the user can correct the error or retry the operation.

IoRaiseInformationalHardError

Notifies the user of an error, providing an I/O error status and an optional string supplying more information.

ExRaiseStatus

Raises an error status and causes a caller-supplied structured exception handler to be called. Useful only to highest-level drivers that supply exception handlers, in particular to file systems.

IoStartNextPacket

Dequeues the next IRP for a given device object, specifies whether the IRP is cancelable, and calls the driver's StartIo routine.

IoStartNextPacketByKey

Dequeues the next IRP for a device object according to a specified sort-key value, specifies whether the IRP is cancelable, and calls the driver's StartIo routine.

IoCompleteRequest

Completes an I/O request, giving a priority boost to the original caller and returning a given IRP to the I/O system for disposal: either to call any IoCompletion routines supplied by higher-level drivers, or to return status to the original requestor of the operation.

IoGetCurrentProcess

Returns a pointer to the current process. Useful only to highest-level drivers.

IoGetInitialStack

Returns the initial base address of the current thread's stack. Useful only to highest-level drivers.

IoGetRemainingStackSize

Returns the amount of available stack space. Useful only to highest-level drivers.

IoGetStackLimits

Returns the boundaries of the current thread's stack frame. Useful only to highest-level drivers.

Driver-Allocated IRPs

IoBuildAsynchronousFsdRequest

Allocates and sets up an IRP that specifies a major function code (IRP_MJ_PNP, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_SHUTDOWN, or IRP_MJ_FLUSH_BUFFERS) with a pointer to:

- The lower driver's device object on which the I/O should occur
- A pointer to a buffer which will contain the data to be read or which contains the data to be written
- The length of the buffer in bytes
- The starting offset on the media
- The I/O status block where the called driver can return status information and the caller's IoCompletion routine can access it

Returns a pointer to the IRP so the caller can set any necessary minor function code and set up its IoCompletion routine before sending the IRP to the target driver.

IoBuildSynchronousFsdRequest

Allocates and sets up an IRP specifying a major function code (IRP_MJ_PNP, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_SHUTDOWN, or IRP_MJ_FLUSH_BUFFERS) with a pointer to:

- The lower driver's device object on which the I/O should occur
- A buffer which will contain the data to be read or which contains the data to be written
- The length of the buffer in bytes
- The starting offset on the media
- An event object to be set to the Signaled state when the requested operation completes
- The I/O status block where the called driver can return status information and the caller's IoCompletion routine can access it.

Returns a pointer to the IRP so the caller can set any necessary minor function code and set up its IoCompletion routine before sending the IRP to the target driver.

IoBuildDeviceIoControlRequest

Allocates and sets up an IRP specifying a major function code (either IRP_MJ_INTERNAL_DEVICE_CONTROL or IRP_MJ_DEVICE_CONTROL) with an optional input or output buffer; a pointer to the lower driver's device object; an event to be set to the Signaled state when the requested operation completes; and an I/O status block to be set by the driver that receives the IRP. Returns a pointer to the IRP so the caller can set the appropriate IOCTL_*XXX* before sending the IRP to the next-lower-level driver.

PoRequestPowerIrp

Allocates and initializes an IRP with major function code IRP_MJ_POWER and then sends the IRP to the top-level driver in the device stack for the specified device object.

IoSizeOfIrp

Returns the size in bytes required for an IRP with a given count of I/O stack locations.

IoAllocateIrp

Allocates an IRP, given the number of I/O stack locations (optionally, for the caller, but at least one for each driver layered under the caller) and whether to charge quota against the caller. Returns a pointer to an IRP in nonpaged system space if successful; otherwise, returns NULL.

IoInitializeIrp

Initializes an IRP, given a pointer to an already allocated IRP, its length in bytes, and its number of I/O stack locations.

IoSetNextIrpStackLocation

Sets the current IRP stack location to the caller's location in an IRP. The stack location must have been allocated by a preceding call to **IoAllocateIrp** that specified a stack-size argument large enough to give the caller its own stack location.

IoAllocateMdl

Allocates an MDL large enough to map the starting address and length supplied by the caller; optionally associates the MDL with a given IRP.

IoBuildPartialMdl

Builds an MDL for the specified starting virtual address and length in bytes from a given source MDL. Drivers that split large transfer requests into a number of smaller transfers can call this routine.

IoFreeMdl

Releases a given MDL allocated by the caller.

IoMakeAssociatedIrp

Allocates and initializes an IRP to be associated with a master IRP sent to the highest-level driver, allowing the driver to "split" the original request and send associated IRPs on to lower-level drivers or to the device.

IoSetCompletionRoutine

Registers a driver-supplied IoCompletion routine with a given IRP, so that the IoCompletion routine is called when lower-level drivers have completed the request. The IoCompletion routine lets the caller release the IRP it allocated with **IoAllocateIrp** or **IoBuildAsynchronousFsdRequest**; to release any other resources it allocated to set up an IRP for lower drivers; and to perform any I/O completion processing necessary.

IoCallDriver

Sends an IRP to a lower-level driver.

IoFreeIrp

Releases an IRP that was allocated by the caller.

IoReuseIrp

Reinitializes for reuse an IRP that was previously allocated by **IoAllocateIrp**.

File Objects

InitializeObjectAttributes

Initializes a parameter of type OBJECT_ATTRIBUTES for a subsequent call to a **ZwCreate***Xxx* or **ZwOpen***Xxx* routine.

ZwCreateFile

Creates or opens a file object representing a physical, logical, or virtual device, a directory, a data file, or a volume.

ZwQueryInformationFile

Returns information about the state or attributes of an open file.

IoGetFileObjectGenericMapping

Returns information about the mapping between generic access rights and specific access rights for file objects.

ZwReadFile

Returns data from an open file.

ZwSetInformationFile

Changes information about the state or attributes of an open file.

ZwWriteFile

Transfers data to an open file.

ZwClose

Releases the handle for an opened object, causing the handle to become invalid and decrementing the reference count of the object handle.

Synchronization

This section describes the kernel-mode support routines that drivers can call to:

- Synchronize the execution of their own standard driver routines
- Temporarily change the current IRQL for a call to a support routine or that return the current IRQL
- Synchronize access to resources with spin locks or to perform interlocked operations without spinlocks
- Manage time-outs or determine system time
- Use system threads or to manage synchronization within a non-arbitrary thread context

Driver Routines and I/O Objects

KeSynchronizeExecution

Synchronizes the execution of a driver-supplied SynchCritSection routine with that of the ISR associated with a set of interrupt objects, given a pointer to the interrupt objects.

IoRequestDpc

Queues a driver-supplied DpcForIsr routine to complete interrupt-driven I/O processing at a lower IRQL.

KeInsertQueueDpc

Queues a DPC to be executed as soon as the IRQL of a processor drops below DISPATCH_LEVEL; returns FALSE if the DPC object is already queued.

KeRemoveQueueDpc

Removes a given DPC object from the DPC queue; returns FALSE if the object is not in the queue.

KeSetImportanceDpc

Controls how a particular DCP is queued and, to some degree, how soon the DPC routine is run.

KeSetTargetProcessorDpc

Controls on which processor a particular DCP subsequently will be queued.

AllocateAdapterChannel

Connects a device object to an adapter object and calls a driver-supplied AdapterControl routine to carry out an I/O operation through the system DMA controller or a busmaster adapter as soon as the appropriate DMA channel and any necessary map registers are available. (This routine reserves exclusive access to a DMA channel and map registers for the specified device.)

FreeAdapterChannel

Releases an adapter object, representing a system DMA channel, and optionally releases map registers, if any were allocated.

FreeMapRegisters

Releases a set of map registers that were saved from a call to **AllocateAdapterChannel**, after the registers have been used by **IoMapTransfer** and the busmaster DMA transfer is complete.

IoAllocateController

Connects a device object to a controller object and calls a driver-supplied ControllerControl routine to carry out an I/O operation on the device controller as soon as the controller is not busy. (This routine reserves exclusive access to the hardware controller for the specified device.)

IoFreeController

Releases a controller object, provided that all device operations queued to the controller for the current IRP have completed.

IoStartTimer

Enables the timer for a given device object and calls the driver-supplied IoTimer routine once per second thereafter.

IoStopTimer

Disables the timer for a given device object so that the driver-supplied IoTimer routine is not called unless the driver re-enables the timer.

KeSetTimer

Sets the absolute or relative interval at which a timer object will be set to the Signaled state and optionally supplies a timer DPC to be executed after the interval expires.

KeSetTimerEx

Sets the absolute or relative interval at which a timer object will be set to the Signaled state, optionally supplies a timer DPC to be executed when the interval expires, and optionally supplies a recurring interval for the timer.

KeCancelTimer

Cancels a timer object before the interval passed to **KeSetTimer** expires; dequeues a timer DPC before the timer interval, if any was set, expires.

KeReadStateTimer

Returns whether a given timer object is set to the Signaled state.

IoStartPacket

Calls the driver's StartIo routine with the given IRP for the given device object or inserts the IRP into the device queue if the device is already busy, specifying whether the IRP is cancelable.

IoStartNextPacket

Dequeues the next IRP for a given device object, specifying whether the IRP is cancelable, and calls the driver's StartIo routine.

IoStartNextPacketByKey

Dequeues the next IRP, according to the specified sort-key value, for a given device object. Specifies whether the IRP is cancelable and calls the driver's StartIo routine.

IoSetCompletionRoutine

Registers a driver-supplied IoCompletion routine with a given IRP, so the IoCompletion routine is called when the next-lower-level driver has completed the requested operation in one or more of the following ways: successfully, with an error, or by canceling the IRP.

IoSetCancelRoutine

Sets or clears the Cancel routine in an IRP. Setting a Cancel routine makes an IRP cancelable.

KeStallExecutionProcessor

Stalls the caller (a device driver) for a given interval on the current processor.

ExAcquireResourceExclusiveLite

Acquires an initialized resource for exclusive access by the calling thread and optionally waits for the resource to be acquired.

ExTryToAcquireResourceExclusiveLite

Acquires a given resource for exclusive access immediately or returns FALSE.

ExAcquireResourceSharedLite

Acquires an initialized resource for shared access by the calling thread and optionally waits for the resource to be acquired.

ExAcquireSharedStarveExclusive

Acquires a given resource for shared access without waiting for any pending attempts to acquire exclusive access to the same resource.

ExAcquireSharedWaitForExclusive

Acquires a given resource for shared access, optionally waiting for any pending exclusive waiters to acquire and release the resource first.

ExReleaseResourceForThreadLite

Releases a given resource that was acquired by the given thread.

ZwReadFile

Reads data from an open file. If the caller opened the file object with certain parameters, the caller can wait on the file handle for completion of the I/O.

ZwWriteFile

Writes data to an open file. If the caller opened the file object with certain parameters, the caller can wait on the file handle for completion of the I/O.

IRQL

KeRaiseIrql

Raises the hardware priority to a given IRQL value, thereby masking off interrupts of equivalent or lower IRQL on the current processor.

KeRaiseIrqlToDpcLevel

Raises the hardware priority to IRQL DISPATCH_LEVEL, thereby masking off interrupts of equivalent or lower IRQL on the current processor.

KeLowerIrql

Restores the IRQL on the current processor to its original value.

KeGetCurrentIrql

Returns the current hardware priority IRQL value.

Spin Locks and Interlocks

IoAcquireCancelSpinLock

Synchronizes cancelable state transitions for IRPs in a multiprocessor-safe manner.

IoSetCancelRoutine

Sets or clears the Cancel routine in an IRP during a cancelable state transition. Setting a Cancel routine makes an IRP cancelable.

IoReleaseCancelSpinLock

Releases the cancel spin lock when the driver has changed the cancelable state of an IRP or releases the cancel spin lock from the driver's Cancel routine.

KeInitializeSpinLock

Initializes a variable of type KSPIN_LOCK, used to synchronize access to data shared among nonISR routines. An initialized spin lock also is a required parameter to the **ExInterlocked***Xxx* routines.

KeAcquireSpinLock

Acquires a spin lock so the caller can synchronize access to shared data safely on multiprocessor platforms.

KeReleaseSpinLock

Releases a spin lock that was acquired by calling **KeAcquireSpinLock** and restores the original IRQL at which the caller was running.

KeAcquireSpinLockAtDpcLevel

Acquires a spin lock, provided that the caller is already running at IRQL DISPATCH_LEVEL.

KeReleaseSpinLockFromDpcLevel

Releases a spin lock that was acquired by calling **KeAcquireSpinLockAtDpcLevel**.

ExInterlocked..List

Insert and remove IRPs in a driver-managed internal queue, which is protected by an initialized spin lock for which the driver provides the storage.

Ke..DeviceQueue

Insert and remove IRPs in a driver-allocated and managed internal device queue object, which is protected by a built-in spin lock.

ExInterlockedAddUlong

Adds a value to a variable of type ULONG as an atomic operation, using a spin lock to ensure multiprocessor-safe access to the variable; returns the value of the variable before the call occurred.

ExInterlockedAddLargeInteger

Adds a value to a variable of type LARGE_INTEGER as an atomic operation, using a spin lock to ensure multiprocessor-safe access to the variable; returns the value of the variable before the call occurred.

InterlockedIncrement

Increments a variable of type LONG as an atomic operation. The sign of the return value is the sign of the result of the operation.

InterlockedDecrement

Decrements a variable of type LONG as an atomic operation. The sign of the return value is the sign of the result of the operation.

InterlockedExchange

Sets a variable of type LONG to a specified value as an atomic operation; returns the value of the variable before the call occurred.

InterlockedExchangeAdd

Adds a value to a given integer variable as an atomic operation; returns the value of the variable before the call occurred.

InterlockedCompareExchange

Compares the values referenced by two pointers. If the values are equal, resets one of the values to a caller-supplied value in an atomic operation.

InterlockedCompareExchangePointer

Compares the pointers referenced by two pointers. If the pointer values are equal, resets one of the values to a caller-supplied value in an atomic operation.

ExInterlockedCompareExchange64

Compares one integer variable to another and, if they are equal, resets the first variable to a caller-supplied ULONGLONG-type value as an atomic operation.

KeGetCurrentProcessorNumber

Returns the current processor number when debugging spin lock usage in SMP machines.

Timers

IoInitializeTimer

Associates a timer with the given device object and registers a driver-supplied IoTimer routine for the device object.

IoStartTimer

Enables the timer for a given device object and calls the driver-supplied IoTimer routine once every second.

IoStopTimer

Disables the timer for a given device object so the driver-supplied IoTimer routine is not called unless the driver re-enables the timer.

KeInitializeDpc

Initializes a DPC object and sets up a driver-supplied CustomTimerDpc routine that can be called with a given context.

KeInitializeTimer

Initializes a notification timer object to the Not-Signaled state.

KeInitializeTimerEx

Initializes a notification or synchronization timer object to the Not-Signaled state.

KeSetTimer

Sets the absolute or relative interval at which a timer object will be set to the Signaled state; optionally supplies a timer DPC to be executed when the interval expires.

KeSetTimerEx

Sets the absolute or relative interval at which a timer object will be set to the Signaled state; optionally supplies a timer DPC to be executed when the interval expires; and optionally supplies a recurring interval for the timer.

KeCancelTimer

Cancels a timer object before the interval passed to **KeSetTimer** expires; dequeues a timer DPC before the timer interval, if any was set, expires.

KeReadStateTimer

Returns TRUE if a given timer object is set to the Signaled state.

KeQuerySystemTime

Returns the current system time.

KeQueryTickCount

Returns the number of interval-timer interrupts that have occurred since the system was booted.

KeQueryTimeIncrement

Returns the number of 100-nanosecond units that are added to the system time at each interval-timer interrupt.

KeQueryInterruptTime

Returns the current value of the system interrupt-time count in 100-nanosecond units.

KeQueryPerformanceCounter

Returns the system performance counter value in hertz.

Driver Threads, Dispatcher Objects, and Resources

KeDelayExecutionThread

Puts the current thread into an alertable or nonalertable wait state for a given interval.

ExInitializeResourceLite

Initializes a resource, for which the caller provides the storage, to be used for synchronization by a set of threads (shared readers, exclusive writers).

ExReinitializeResourceLite

Reinitializes an existing resource variable.

ExAcquireResourceExclusiveLite

Acquires an initialized resource for exclusive access by the calling thread and optionally waits for the resource to be acquired.

ExTryToAcquireResourceExclusiveLite

Either acquires a given resource for exclusive access immediately, or returns FALSE.

ExAcquireResourceSharedLite

Acquires an initialized resource for shared access by the calling thread and optionally waits for the resource to be acquired.

ExAcquireSharedStarveExclusive

Acquires a given resource for shared access without waiting for any pending attempts to acquire exclusive access to the same resource.

ExAcquireSharedWaitForExclusive

Acquires a given resource for shared access, optionally waiting for any pending exclusive waiters to acquire and release the resource first.

ExIsResourceAcquiredExclusiveLite

Returns whether the calling thread has exclusive access to a given resource.

ExIsResourceAcquiredSharedLite

Returns how many times the calling thread has acquired shared access to a given resource.

ExGetExclusiveWaiterCount

Returns the number of threads currently waiting to acquire a given resource for exclusive access.

ExGetSharedWaiterCount

Returns the number of threads currently waiting to acquire a given resource for shared access.

ExConvertExclusiveToSharedLite

Converts a given resource from acquired for exclusive access to acquired for shared access.

ExGetCurrentResourceThread

Returns the thread ID of the current thread.

ExReleaseResourceForThreadLite

Releases a given resource that was acquired by the given thread.

ExDeleteResourceLite

Deletes a caller-initialized resource from the system's resource list.

IoQueueWorkItem

Queues an initialized work queue item so the driver-supplied routine will be called when a system worker thread is given control.

KeSetTimer

Sets the absolute or relative interval at which a timer object will be set to the Signaled state, and optionally supplies a timer DPC to be executed when the interval expires.

KeSetTimerEx

Sets the absolute or relative interval at which a timer object will be set to the Signaled state. Optionally supplies a timer DPC to be executed when the interval expires and a recurring interval for the timer.

KeCancelTimer

Cancels a timer object before the interval passed to **KeSetTimer** expires. Dequeues a timer DPC before the timer interval (if any) expires.

KeReadStateTimer

Returns TRUE if a given timer object is set to the Signaled state.

KeSetEvent

Returns the previous state of a given event object and sets the event (if not already Signaled) to the Signaled state.

KeClearEvent

Resets an event to the Not-Signaled state.

KeResetEvent

Returns the previous state of an event object and resets the event to the Not-Signaled state.

KeReadStateEvent

Returns the current state (nonzero for Signaled or zero for Not-Signaled) of a given event object.

ExAcquireFastMutex

Acquires an initialized fast mutex, possibly after putting the caller into a wait state until it is acquired, and gives the calling thread ownership with APCs disabled.

ExTryToAcquireFastMutex

Acquires the given fast mutex immediately for the caller with APCs disabled, or returns FALSE.

ExReleaseFastMutex

Releases ownership of a fast mutex that was acquired with **ExAcquireFastMutex** or **ExTryToAcquireFastMutex**.

ExAcquireFastMutexUnsafe

Acquires an initialized fast mutex, possibly after putting the caller into a wait state until it is acquired.

ExReleaseFastMutexUnsafe

Releases ownership of a fast mutex that was acquired with **ExAcquireFastMutexUnsafe**.

KeReleaseMutex

Releases a given mutex object, specifying whether the caller will call one of the **KeWait***Xxx* routines as soon as **KeReleaseMutex** returns the previous value of the mutex state (a zero for Signaled; otherwise, Not-Signaled).

KeReadStateMutex

Returns the current state (one for Signaled or any other value for Not-Signaled) of a given mutex object.

KeReleaseSemaphore

Releases a given semaphore object. Supplies a (run-time) priority boost for waiting threads if the release sets the semaphore to the Signaled state. Augments the semaphore count by a given value and specifies whether the caller will call one of the **KeWait***Xxx* routines as soon as **KeReleaseSemaphore** returns.

KeReadStateSemaphore

Returns the current state (zero for Not-Signaled or a positive value for Signaled) of a given semaphore object.

KeWaitForSingleObject

Puts the current thread into an alertable or nonalertable wait state until a given dispatcher object is set to the Signaled state or (optionally) until the wait times out.

KeWaitForMutexObject

Puts the current thread into an alertable or nonalertable wait state until a given mutex is set to the Signaled state or (optionally) until the wait times out.

KeWaitForMultipleObjects

Puts the current thread into an alertable or nonalertable wait state until any one or all of a number of dispatcher objects are set to the Signaled state or (optionally) until the wait times out.

PsGetCurrentThread

Returns a handle for the current thread.

KeGetCurrentThread

Returns a pointer to the opaque thread object that represents the current thread.

IoGetCurrentProcess

Returns a handle for the process of the current thread.

PsGetCurrentProcess

Returns a pointer to the process of the current thread.

KeEnterCriticalRegion

Temporarily disables the delivery of normal kernel APCs while a highest-level driver is running in the context of the user-mode thread that requested the current I/O operation. Special kernel-mode APCs are still delivered.

KeLeaveCriticalRegion

Re-enables, as soon as possible, the delivery of normal kernel-mode APCs that were disabled by a preceding call to **KeEnterCriticalRegion**.

KeSaveFloatingPointState

Saves the current thread's nonvolatile floating-point context so that the caller can carry out its own floating-point operations.

KeRestoreFloatingPointState

Restores the previous nonvolatile floating-point context that was saved with **KeSaveFloatingPointState**.

ZwSetInformationThread

Sets the priority of a given thread for which the caller has a handle.

PsGetCurrentProcessId

Returns the system-assigned identifier of the current process.

PsGetCurrentThreadId

Returns the system-assigned identifier of the current thread.

PsSetCreateProcessNotifyRoutine

Registers a highest level driver's callback that is subsequently notified whenever a new process is created or existing process deleted.

PsSetCreateThreadNotifyRoutine

Registers a highest level driver's callback that is subsequently notified whenever a new thread is created or an existing thread is deleted.

PsSetLoadImageNotifyRoutine

Registers a callback routine for a highest level system-profiling driver. The callback is subsequently notified whenever a new image is loaded for execution.

Memory

This section describes the kernel-mode support routines and macros that drivers can call to:

- Allocate and free temporary buffers

- Allocate long-term internal driver buffers
- Manage buffered data or to initialize driver-allocated buffers
- Get mapped addresses and to allocate or manage MDLs (memory descriptor lists)
- Manipulate buffers and MDLs
- Communicate with their respective devices
- Lock and unlock their pageable code or data sections, or that they can call to make their entire driver pageable
- Set up mapped sections and views of memory

Buffer Management

ExAllocatePool

Allocates (optionally cache-aligned) memory from paged or nonpaged system space.

ExAllocatePoolWithQuota

Allocates pool memory charging quota against the original requestor of the I/O operation. (Only highest-level drivers can call this routine.)

ExAllocatePoolWithTag

Allocates (optionally cache-aligned) tagged memory from paged or nonpaged system space. The caller-supplied tag is put into any crash dump of memory that occurs.

ExAllocatePoolWithQuotaTag

Allocates tagged pool memory charging quota against the original requestor of the I/O operation. The caller-supplied tag is put into any crash dump of memory that occurs. Only highest-level drivers can call this routine.

ExFreePool

Releases memory to paged or nonpaged system space.

ExInitializeNPagedLookasideList

Initializes a lookaside list of nonpaged memory. After successful initialization of the list, fixed-size blocks can be allocated from, and freed to, the lookaside list.

ExAllocateFromNPagedLookasideList

Removes the first entry from the specified lookaside list in nonpaged memory. If the lookaside list is empty, allocates an entry from nonpaged pool.

ExFreeToNPagedLookasideList

Returns an entry to the specified lookaside list in nonpaged memory. If the list has reached its maximum size, returns the entry to nonpaged pool.

ExDeleteNPagedLookasideList

Deletes a nonpaged lookaside list.

ExInitializePagedLookasideList

Initializes a lookaside list of paged memory. After successful initialization of the list, fixed-size blocks can be allocated from and freed to the lookaside list.

ExAllocateFromPagedLookasideList

Removes the first entry from the specified lookaside list in paged memory. If the lookaside list is empty, allocates an entry from paged pool.

ExFreeToPagedLookasideList

Returns an entry to the specified lookaside list in paged memory. If the list has reached its maximum size, returns the entry to paged pool.

ExDeletePagedLookasideList

Deletes a paged lookaside list.

MmQuerySystemSize

Returns an estimate (small, medium, or large) of the amount of memory available on the current platform.

MmIsThisAnNtAsSystem

Returns TRUE if the machine is running as a Windows NT/Windows 2000 server. If this routine returns TRUE, the caller is likely to require more resources to process I/O requests, and the machine is a server so it is likely to have more resources available.

Long-Term Internal Driver Buffers

MmAllocateContiguousMemory

Allocates a range of physically contiguous, cache-aligned memory in nonpaged pool.

MmFreeContiguousMemory

Releases a range of physically contiguous memory when the driver unloads.

MmAllocateNonCachedMemory

Allocates a virtual address range of noncached and cache-aligned memory in nonpaged system space (pool).

MmFreeNonCachedMemory

Releases a virtual address range of noncached memory in nonpaged system space when the driver unloads.

AllocateCommonBuffer

Allocates and maps a logically contiguous region of memory that is simultaneously accessible both from the processor and from a device, given access to an adapter object, the requested length of the memory region to allocate, and access to variables where the starting logical and virtual addresses of the allocated region are returned. Returns TRUE if the requested length was allocated. Can be used for continuous busmaster DMA or for system DMA using the autoinitialize mode of a system DMA controller.

FreeCommonBuffer

Releases an allocated common buffer and unmaps it, given access to the adapter object, the length, and the starting logical and virtual addresses of the region to be freed when the driver unloads. Arguments must match those passed in the call to **AllocateCommonBuffer**.

Buffered Data and Buffer Initialization

RtlCompareMemory

Compares data, given pointers to caller-supplied buffers and the length in bytes for the comparison. Returns the number of bytes that are equal.

RtlCopyMemory

Copies the data from one caller-supplied buffer to another, given pointers to both buffers and the length in bytes to be copied.

RtlMoveMemory

Copies the data from one caller-supplied memory range to another, given pointers to the base of both ranges and the length in bytes to be copied.

RtlFillMemory

Fills a caller-supplied buffer with the specified UCHAR value, given a pointer to the buffer and the length in bytes to be filled.

RtlZeroMemory

Fills a buffer with zeros, given a pointer to the caller-supplied buffer and the length in bytes to be filled.

RtlStoreUshort

Stores a USHORT value at a given address, avoiding alignment faults.

RtlRetrieveUshort

Retrieves a USHORT value at a given address, avoiding alignment faults, and stores the value at a given address, that is assumed to be aligned.

RtlStoreUlong

Stores a ULONG value at a given address, avoiding alignment faults.

RtlRetrieveUlong

Retrieves a ULONG value at a given address, avoiding alignment faults, and stores the value at a given address, that is assumed to be aligned.

Address Mappings and MDLs

MmGetPhysicalAddress

Returns the corresponding physical address for a given valid virtual address.

MmGetMdlVirtualAddress

Returns a (possibly invalid) virtual address for a buffer described by a given MDL; the returned address, used as an index to a physical address entry in the MDL, can be input to **MapTransfer** for drivers that use DMA.

MmGetSystemAddressForMdl

Returns a system-space virtual address that maps the physical pages described by a given MDL for drivers whose devices must use PIO. If no virtual address exists, one is assigned. If none are available, a bug check is issued. Windows 2000 drivers should use **MmGetSystemAddressForMdlSafe** instead.

MmGetSystemAddressForMdlSafe

Returns a system-space virtual address that maps the physical pages described by a given MDL for drivers whose devices must use PIO. If no virtual address exists, one is assigned.

MmBuildMdlForNonPagedPool

Fills in the corresponding physical addresses of a given MDL that specifies a range of virtual addresses in nonpaged pool.

MmGetMdlByteCount

Returns the length in bytes of the buffer mapped by a given MDL.

MmGetMdlByteOffset

Returns the byte offset within a page of the buffer described by a given MDL.

MmMapLockedPages

Maps already locked physical pages, described by a given MDL, to a returned virtual address range.

MmUnmapLockedPages

Releases a mapping set up by **MmMapLockedPages**.

MmIsAddressValid

Returns whether a page fault will occur if a read or write operation is done at the given virtual address.

MmSizeOfMdl

Returns the number of bytes required for an MDL describing the buffer specified by the given virtual address and length in bytes.

MmCreateMdl

Allocates and initializes an MDL describing a buffer specified by the given virtual address and length in bytes; returns a pointer to the MDL.

MmPrepareMdlForReuse

Reinitializes a caller-created MDL for reuse.

MmInitializeMdl

Initializes a caller-created MDL to describe a buffer specified by the given virtual address and length in bytes.

MmMapIoSpace

Maps a physical address range to a cached or noncached virtual address range in nonpaged system space.

MmUnmapIoSpace

Unmaps a virtual address range from a physical address range.

MmProbeAndLockPages

Probes the pages specified in an MDL for a particular kind of access, makes the pages resident, and locks them in memory; returns the MDL updated with corresponding physical addresses. (Usually, only highest-level drivers call this routine.)

MmUnlockPages

Unlocks the previously probed and locked pages specified in an MDL.

IoAllocateMdl

Allocates an MDL large enough to map the starting address and length supplied by the caller; optionally associates the MDL with a given IRP.

IoBuildPartialMdl

Builds an MDL for the specified starting virtual address and length in bytes from a given source MDL. Drivers that split large transfer requests into a number of smaller transfers can call this routine.

IoFreeMdl

Releases a given MDL allocated by the caller.

Buffer and MDL Management

ADDRESS_AND_SIZE_TO_SPAN_PAGES

Returns the number of pages required to contain a given virtual address and size in bytes.

BYTE_OFFSET

Returns the byte offset of a given virtual address within the page.

BYTES_TO_PAGES

Returns the number of pages necessary to contain a given number of bytes.

PAGE_ALIGN

Returns the page-aligned virtual address for the page that contains a given virtual address.

ROUND_TO_PAGES

Rounds a given size in bytes up to a page-size multiple.

Device Memory Access

For the following, *XXX*_**REGISTER**_*XXX* indicates device memory that is mapped onto system space, while *XXX*_**PORT**_*XXX* indicates device memory in I/O space.

READ_PORT_UCHAR

Reads a UCHAR value from the given I/O port address.

READ_PORT_USHORT

Reads a USHORT value from the given I/O port address.

READ_PORT_ULONG

Reads a ULONG value from the given I/O port address.

READ_PORT_BUFFER_UCHAR

Reads a given count of UCHAR values from the given I/O port into a given buffer.

READ_PORT_BUFFER_USHORT

Reads a given count of USHORT values from the given I/O port into a given buffer.

READ_PORT_BUFFER_ULONG

Reads a given count of ULONG values from the given I/O port into a given buffer.

WRITE_PORT_UCHAR

Writes a given UCHAR value to the given I/O port address.

WRITE_PORT_USHORT

Writes a given USHORT value to the given I/O port address.

WRITE_PORT_ULONG

Writes a given ULONG value to the given I/O port address.

WRITE_PORT_BUFFER_UCHAR

Writes a given count of UCHAR values from a given buffer to the given I/O port.

WRITE_PORT_BUFFER_USHORT

Writes a given count of USHORT values from a given buffer to the given I/O port.

WRITE_PORT_BUFFER_ULONG

Writes a given count of ULONG values from a given buffer to the given I/O port.

READ_REGISTER_UCHAR

Reads a UCHAR value from the given register address in memory space.

READ_REGISTER_USHORT

Reads a USHORT value from the given register address in memory space.

READ_REGISTER_ULONG

Reads a ULONG value from the given register address in memory space.

READ_REGISTER_BUFFER_UCHAR

Reads a given count of UCHAR values from the given register address into the given buffer.

READ_REGISTER_BUFFER_USHORT

Reads a given count of USHORT values from the given register address into the given buffer.

READ_REGISTER_BUFFER_ULONG

Reads a given count of ULONG values from the given register address into the given buffer.

WRITE_REGISTER_UCHAR

Writes a given UCHAR value to the given register address in memory space.

WRITE_REGISTER_USHORT

Writes a given USHORT value to the given register address in memory space.

WRITE_REGISTER_ULONG

Writes a given ULONG value to the given register address in memory space.

WRITE_REGISTER_BUFFER_UCHAR

Writes a given count of UCHAR values from a given buffer to the given register address.

WRITE_REGISTER_BUFFER_USHORT

Writes a given count of USHORT values from a given buffer to the given register address.

WRITE_REGISTER_BUFFER_ULONG

Writes a given count of ULONG values from a given buffer to the given register address.

Pageable Drivers

MmLockPagableCodeSection

Locks a set of driver routines marked with a special compiler directive into system space.

MmLockPagableDataSection

Locks data marked with a special compiler directive into system space, when that data is accessed infrequently, predictably, and at an IRQL less than DISPATCH_LEVEL.

MmLockPagableSectionByHandle

Locks a pageable section into system memory using a handle returned from **MmLockPagableCodeSection** or **MmLockPagableDataSection**.

MmUnlockPagableImageSection

Releases a section that was previously locked into system space when the driver is no longer processing IRPs, or when the contents of the section is no longer required.

MmPageEntireDriver

Lets a driver page all of its code and data regardless of the attributes of the various sections in the driver's image.

MmResetDriverPaging

Resets a driver's pageable status to that specified by the sections which make up the driver's image.

Sections and Views

InitializeObjectAttributes

Sets up a parameter of type OBJECT_ATTRIBUTES for a subsequent call to a **ZwCreate***Xxx* or **ZwOpen***Xxx* routine.

ZwOpenSection

Obtains a handle for an existing section, provided that the requested access can be allowed.

ZwMapViewOfSection

Maps a view of an open section into the virtual address space of a process. Returns an offset into the section (base of the mapped view) and the size mapped.

ZwUnMapViewOfSection

Releases a mapped view in the virtual address space of a process.

Access to Structures

ARGUMENT_PRESENT

Returns FALSE if an argument pointer is NULL; otherwise returns TRUE.

CONTAINING_RECORD

Returns the base address of an instance of a structure given the structure type and the address of a field within it.

FIELD_OFFSET

Returns the byte offset of a named field in a known structure type.

DMA

IoGetDmaAdapter

Returns a pointer to an adapter object that represents either the DMA channel to which the driver's device is connected or the driver's busmaster adapter. Also returns the maximum number of map registers the driver can specify for each DMA transfer.

MmGetMdlVirtualAddress

Returns the base virtual address of a buffer described by a given MDL. The returned address, used as an index to a physical address entry in the MDL, can be input to **MapTransfer**.

MmGetSystemAddressForMdlSafe

Returns a nonpaged system-space virtual address for the base of the memory area described by an MDL. It maps the physical pages described by the MDL into system space, if they are not already mapped to system space. WDM drivers should use **MmGetSystemAddressForMdl** instead.

ADDRESS_AND_SIZE_TO_SPAN_PAGES

Returns the number of pages spanned by the virtual range defined by a virtual address and a length in bytes. A driver can use this macro to determine whether a transfer request must be split into partial transfers.

AllocateAdapterChannel

Reserves exclusive access to a DMA channel and map registers for a device. When the channel and registers are available, this routine calls a driver-supplied AdapterControl routine to carry out an I/O operation through either the system DMA controller or a busmaster adapter.

AllocateCommonBuffer

Allocates and maps a logically contiguous region of memory that is simultaneously accessible from both the processor and a device. This routine returns TRUE if the requested length was allocated.

FlushAdapterBuffers

Forces any data remaining in either a busmaster adapter's or the system DMA controller's internal buffers to be written into memory or to the device.

FreeAdapterChannel

Releases an adapter object that represents a system DMA channel, and optionally releases any allocated map registers.

FreeCommonBuffer

Releases and unmaps a previously allocated common buffer. Arguments must match those passed in an earlier call to **AllocateCommonBuffer**.

FreeMapRegisters

Releases a set of map registers that were saved from a call to **AllocateAdapterChannel**. A driver calls this routine after using the registers in one or more calls to **MapTransfer**, flushing the cache by calling **FlushAdapterBuffers**, and completing the busmaster DMA transfer.

GetDmaAlignment

Returns the buffer alignment requirements for a DMA controller or device.

GetScatterGatherList

Prepares the system for scatter/gather DMA for a device and calls a driver-supplied routine to carry out the I/O operation. For devices that support scatter/gather DMA, this routine combines the functionality of **AllocateAdapterChannel** and **MapTransfer**.

KeFlushIoBuffers

Flushes the memory region described by an MDL from all processors' caches into memory.

MapTransfer

Sets up map registers for an adapter object previously allocated by **AllocateAdapter-Channel** to map a transfer from a locked-down buffer. Returns the logical address of the mapped region and, for busmaster devices that support scatter/gather, the number of bytes mapped.

PutDmaAdapter

Frees an adapter object previously allocated by **IoGetDmaAdapter**.

PutScatterGatherList

Frees map registers and scatter/gather list previously allocated by **GetScatterGatherList**.

ReadDmaCounter

Returns the number of bytes yet to be transferred during the current system DMA operation (in autoinitialize mode).

PIO

MmProbeAndLockPages

Probes the pages specified in an MDL for a particular kind of access, makes the pages resident, and locks them in memory; returns the MDL updated with corresponding physical addresses.

MmGetSystemAddressForMdlSafe

Returns a system-space virtual address that maps the physical pages described by a given MDL for drivers whose devices must use PIO. If no virtual address exists, one is assigned. Windows 98 drivers should use **MmGetSystemAddressForMdl** instead.

KeFlushIoBuffers

Flushes the memory region described by a given MDL from all processors' caches into memory.

MmUnlockPages

Unlocks the previously probed and locked pages specified in an MDL.

MmMapIoSpace

Maps a physical address range to a cached or noncached virtual address range in nonpaged system space.

MmUnmapIoSpace

Unmaps a virtual address range from a physical address range.

Driver-Managed Queues

KeInitializeSpinLock

Initializes a variable of type KSPIN_LOCK. An initialized spin lock is a required parameter to the **Ex..InterlockedList** routines.

InitializeListHead

Sets up a queue header for a driver's internal queue, given a pointer to driver-supplied storage for the queue header and queue. An initialized queue header is a required parameter to the **ExInterlockedInsert/Remove..List** routines.

ExInterlockedInsertTailList

Inserts an entry at the tail of a doubly-linked list, using a spin lock to ensure multiprocessor-safe access to the list and atomic modification of the list links.

ExInterlockedInsertHeadList

Inserts an entry at the head of a doubly-linked list, using a spin lock to ensure multiprocessor-safe access to the list and atomic modification of the links in the list.

ExInterlockedRemoveHeadList

Removes an entry from the head of a doubly-linked list, using a spin lock to ensure multiprocessor-safe access to the list and atomic modification of the links in the list.

ExInterlockedPopEntryList

Removes an entry from the head of a singly-linked list as an atomic operation, using a spin lock to ensure multiprocessor-safe access to the list.

ExInterlockedPushEntryList

Inserts an entry at the head of a singly-linked list as an atomic operation, using a spin lock to ensure multiprocessor-safe access to the list.

IsListEmpty

Returns TRUE if a queue is empty. (This type of doubly-linked list is not protected by a spin lock, unless the caller explicitly manages synchronization to queued entries with an initialized spin lock for which the caller supplies the storage.)

InsertTailList

Queues an entry at the end of the list.

InsertHeadList

Queues an entry at the head of the list.

RemoveHeadList

Dequeues an entry at the head of the list.

RemoveTailList

Dequeues an entry at the end of the list.

RemoveEntryList

Returns whether a given entry is in the given list and dequeues the entry if it is.

PushEntryList

Inserts an entry into the queue. (This type of singly-linked list is not protected by a spin lock, unless the caller explicitly manages synchronization to queued entries with an initialized spin lock for which the caller supplies the storage.)

PopEntryList

Removes an entry from the queue.

ExInterlockedPopEntrySList

Removes an entry from the head of a sequenced, singly-linked list that was set up with **ExInitializeSListHead**.

ExInterlockedPushEntrySList

Queues an entry at the head of a sequenced, singly-linked list that was set up with **ExInitializeSListHead**.

ExQueryDepthSList

Returns the number of entries currently queued in a sequenced, singly-linked list.

ExInitializeNPagedLookasideList

Sets up a lookaside list, protected by a system-supplied spin lock, in nonpaged pool from which the driver can allocate and free blocks of a fixed size.

KeInitializeDeviceQueue

Initializes a device queue object to a not-busy state, setting up an associated spin lock for multiprocessor-safe access to device queue entries.

KeInsertDeviceQueue

Acquires the device queue spin lock and queues an entry to a device driver if the device queue is not empty; otherwise, inserts the entry at the tail of the device queue.

KeInsertByKeyDeviceQueue

Acquires the device queue spin lock and queues an entry to a device driver if the device queue is not empty; otherwise, inserts the entry into the queue according to the given sort-key value.

KeRemoveDeviceQueue

Removes an entry from the head of a given device queue.

KeRemoveByKeyDeviceQueue

Removes an entry, selected according to the specified sort-key value, from the given device queue.

KeRemoveEntryDeviceQueue

Determines whether a given entry is in the given device queue and, if so, dequeues the entry.

Driver System Threads

PsCreateSystemThread

Creates a kernel-mode thread associated with a given process object or with the default system process. Returns a handle for the thread.

PsTerminateSystemThread

Terminates the current thread and satisfies as many waits as possible for the current thread object.

PsGetCurrentThread

Returns a handle for the current thread.

KeGetCurrentThread

Returns a pointer to the opaque thread object that represents the current thread.

KeQueryPriorityThread

Returns the current priority of a given thread.

KeSetBasePriorityThread

Sets up the run-time priority, relative to the system process, for a driver-created thread.

KeSetPriorityThread

Sets up the run-time priority for a driver-created thread with a real-time priority attribute.

KeDelayExecutionThread

Puts the current thread into an alertable or nonalertable wait state for a given interval.

IoQueueWorkItem

Queues an initialized work queue item so the driver-supplied routine will be called when a system worker thread is given control.

ZwSetInformationThread

Sets the priority of a given thread for which the caller has a handle.

Strings

RtlInitString

Initializes the specified string in a buffer.

RtlInitAnsiString

Initializes the specified ANSI string in a buffer.

RtlInitUnicodeString

Initializes the specified Unicode string in a buffer.

RtlAnsiStringToUnicodeSize

Returns the size in bytes required to hold a Unicode version of a given buffered ANSI string.

RtlAnsiStringToUnicodeString

Converts a buffered ANSI string to a Unicode string, given a pointer to the source-string buffer and the address of caller-supplied storage for a pointer to the destination buffer. (This routine allocates a destination buffer if the caller does not supply the storage.) You can also use the string manipulation routines provided by a compiler to convert ANSI strings to Unicode.

RtlFreeUnicodeString

Releases a buffer containing a Unicode string, given a pointer to the buffer returned by **RtlAnsiStringToUnicodeString**.

RtlUnicodeStringToAnsiString

Converts a buffered Unicode string to an ANSI string, given a pointer to the source-string buffer and the address of caller-supplied storage for a pointer to the destination buffer. (This routine allocates a destination buffer if the caller does not supply the storage.)

RtlFreeAnsiString

Releases a buffer containing an ANSI string, given a pointer to the buffer returned by **RtlUnicodeStringToAnsiString**.

RtlAppendUnicodeStringToString

Concatenates a copy of a buffered Unicode string with a buffered Unicode string, given pointers to both buffers.

RtlAppendUnicodeToString

Concatenates a given input string with a buffered Unicode string, given a pointer to the buffer.

RtlCopyString

Copies the source string to the destination, given pointers to both buffers, or sets the length of the destination string (but not the length of the destination buffer) to zero if the optional pointer to the source-string buffer is NULL.

RtlCopyUnicodeString

Copies the source string to the destination, given pointers to both buffers, or sets the length of the destination string (but not the length of the destination buffer) to zero if the optional pointer to the source-string buffer is NULL.

RtlEqualString

Returns TRUE if the given ANSI alphabetic strings are equivalent.

RtlEqualUnicodeString

Returns TRUE if the given buffered strings are equivalent.

RtlCompareString

Compares two buffered, single-byte character strings and returns a signed value indicating whether they are equivalent or which is greater.

RtlCompareUnicodeString

Compares two buffered Unicode strings and returns a signed value indicating whether they are equivalent or which is greater.

RtlUpperString

Converts a copy of a buffered string to uppercase and stores the copy in a destination buffer.

RtlUpcaseUnicodeString

Converts a copy of a buffered Unicode string to uppercase and stores the copy in a destination buffer.

RtlIntegerToUnicodeString

Converts an unsigned integer value in the specified base to one or more Unicode characters in a buffer.

RtlUnicodeStringToInteger

RtlUnicodeStringToInteger converts the Unicode string representation of an integer into its integer equivalent.

Data Conversions

InterlockedExchange

Sets a variable of type LONG to a given value as an atomic operation; returns the original value of the variable.

RtlConvertLongToLargeInteger

Converts a given LONG value to a LARGE_INTEGER value.

RtlConvertUlongToLargeInteger

Converts a given ULONG value to a LARGE_INTEGER value.

RtlTimeFieldsToTime

Converts information in a TIME_FIELDS structure to system time.

RtlTimeToTimeFields

Converts a system time value into a buffered TIME_FIELDS value.

ExSystemTimeToLocalTime

Adds the time-zone bias for the current locale to GMT system time, converting it to local time.

ExLocalTimeToSystemTime

Subtracts the time-zone bias from the local time, converting it to GMT system time.

RtlAnsiStringToUnicodeString

Converts a buffered ANSI string to a Unicode string, given a pointer to the source-string buffer and the address of caller-supplied storage for a pointer to the destination buffer. (This routine allocates a destination buffer if the caller does not supply the storage.)

RtlUnicodeStringToAnsiString

Converts a buffered Unicode string to an ANSI string, given a pointer to the source-string buffer and the address of caller-supplied storage for a pointer to the destination buffer. (This routine allocates a destination buffer if the caller does not supply the storage.)

RtlUpperString

Converts a copy of a buffered string to uppercase and stores the copy in a destination buffer.

RtlUpcaseUnicodeString

Converts a copy of a buffered Unicode string to uppercase and stores the copy in a destination buffer.

RtlCharToInteger

Converts a single-byte character value into an integer in the specified base.

RtlIntegerToUnicodeString

Converts an unsigned integer value in the specified base to one or more Unicode characters in the given buffer.

RtlUnicodeStringToInteger

Converts a Unicode string representation of an integer into its integer equivalent.

Access to Driver-Managed Objects

ExCreateCallback

Creates or opens a callback object.

ExNotifyCallback

Calls the callback routines registered with a previously created or opened callback object.

ExRegisterCallback

Registers a callback routine with a previously created or opened callback object, so that the caller can be notified when conditions defined for the callback occur.

ExUnregisterCallback

Cancels the registration of a callback routine with a callback object.

IoRegisterDeviceInterface

Registers device functionality (a device interface) that a driver can enable for use by applications or other system components.

IoSetDeviceInterfaceState

Enables or disables a previously registered device interface. Applications and other system components can open only interfaces that are enabled.

IoGetDeviceInterfaceAlias

Returns the alias device interface of the specified interface class, if the alias exists. Device interfaces are considered aliases if they are exposed by the same underlying device and have identical interface reference strings, but are of different interface classes.

IoGetDeviceInterfaces

Returns a list of device interfaces of a particular device interface class (such as all devices on the system that support a HID interface).

IoGetFileObjectGenericMapping

Returns information about the mapping between generic access rights and specific access rights for file objects.

IoSetShareAccess

Sets the access allowed to a given file object representing a device. (Only highest-level drivers can call this routine.)

IoCheckShareAccess

Checks whether a request to open a file object specifies a desired access that is compatible with the current shared access permissions for the open file object. (Only highest-level drivers can call this routine.)

IoUpdateShareAccess

Modifies the current share-access permissions on the given file object. (Only highest-level drivers can call this routine.)

IoRemoveShareAccess

Restores the shared-access permissions on the given file object that were modified by a preceding call to **IoUpdateShareAccess**.

RtlLengthSecurityDescriptor

Returns the size in bytes of a given security descriptor.

RtlValidSecurityDescriptor

Returns whether a given security descriptor is valid.

RtlCreateSecurityDescriptor

Initializes a new security descriptor to an absolute format with default values (in effect, with no security constraints).

RtlSetDaclSecurityDescriptor

Sets the discretionary ACL information for a given security descriptor in absolute format.

SeAssignSecurity

Builds a security descriptor for a new object, given the security descriptor of its parent directory (if any) and an originally requested security for the object.

SeDeassignSecurity

Deallocates the memory associated with a security descriptor that was created with **SeAssignSecurity**.

SeValidSecurityDescriptor

Returns whether a given security descriptor is structurally valid.

SeAccessCheck

Returns a Boolean indicating whether the requested access rights can be granted to an object protected by a security descriptor and, possibly, a current owner.

SeSinglePrivilegeCheck

Returns a Boolean indicating whether the current thread has at least the given privilege level.

Error Handling

IoAllocateErrorLogEntry

Allocates and initializes an error log packet; returns a pointer so the caller can supply error-log data and call **IoWriteErrorLogEntry** with the packet.

IoWriteErrorLogEntry

Queues a previously allocated error log packet, filled in by the driver, to the system error logging thread.

IoIsErrorUserInduced

Returns a Boolean indicating whether an I/O request failed due to one of the following (user-correctable) conditions: STATUS_IO_TIMEOUT, STATUS_DEVICE_NOT_READY, STATUS_UNRECOGNIZED_MEDIA, STATUS_VERIFY_REQUIRED, STATUS_WRONG_VOLUME, STATUS_MEDIA_WRITE_PROTECTED, or STATUS_NO_MEDIA_IN_DEVICE. If the result is TRUE, a removable-media driver must call **IoSetHardErrorOrVerifyDevice** before completing the IRP.

IoSetHardErrorOrVerifyDevice

Supplies the device object for which the given IRP was failed due to a user-induced error, such as supplying the incorrect media for the requested operation or changing the media before the requested operation was completed. (A file system driver uses the associated

device object to send a popup to the user; the user can then correct the error or retry the operation.)

IoSetThreadHardErrorMode

Enables or disables error reporting for the current thread using **IoRaiseHardError** or **IoRaiseInformationalHardError**.

IoGetDeviceToVerify

Returns a pointer to the device object, representing a removable-media device, that is the target of the given thread's I/O request. (This routine is useful only to file systems or other highest-level drivers.)

IoRaiseHardError

Causes a popup to be sent to the user indicating that the given IRP was failed on the given device object for an optional VPB, so that the user can correct the error or retry the operation.

IoRaiseInformationalHardError

Causes a popup to be sent to the user, showing an I/O error status and optional string supplying more information.

ExRaiseStatus

Raises an error status so that a caller-supplied structured exception handler is called. (This routine is useful only to highest-level drivers that supply exception handlers, in particular to file systems.)

KeBugCheckEx

Brings down the system in a controlled manner, displaying the bugcheck code and possibly more information, after the caller discovers an unrecoverable inconsistency that will corrupt the system unless it is brought down. After the system is brought down, this routine displays bug-check and possibly other information. (This routine can be called when debugging under-development drivers. Otherwise, drivers should never call this routine when they can handle an error by failing an IRP and by calling **IoAllocateErrorLogEntry** and **IoWrite-ErrorLogEntry**.)

KeBugCheck

Brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that will corrupt the system if the caller continues to run. **KeBugCheckEx** is preferable.

KeInitializeCallbackRecord

Initializes a bug-check callback record before a device driver calls **KeRegisterBugCheck-Callback**.

KeRegisterBugCheckCallback

Registers the device driver's bug-check callback routine, that is called if a system bug check occurs. Such a driver-supplied routine saves driver-determined state information, such as the contents of device registers, that would not otherwise be written into the system crash-dump file.

KeDeregisterBugCheckCallback

Removes a device driver's callback routine from the set of registered bug-check callbacks.

CHAPTER 2

Executive Support Routines

References for the **Ex***Xxx* routines are in alphabetical order.

For an overview of the functionality of these routines, see Chapter 1, *Summary of Kernel-Mode Support Routines*.

ExAcquireFastMutex

```
VOID
  ExAcquireFastMutex(
  IN PFAST_MUTEX  FastMutex
  );
```

The **ExAcquireFastMutex** support routine acquires the given fast mutex with APCs to the current thread disabled.

Parameters

FastMutex

Pointer to an initialized fast mutex for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Comments

ExAcquireFastMutex puts the caller into a wait state if the given fast mutex cannot be acquired immediately. Otherwise, the caller is given ownership of the fast mutex with APCs to the current thread disabled until it releases the fast mutex.

Use **ExTryToAcquireFastMutex** if the current thread can do other work before it waits on the acquisition of the given mutex.

Any fast mutex acquired using **ExAcquireFastMutex** or **ExTryToAcquireFastMutex** must be released with **ExReleaseFastMutex**.

Callers of **ExAcquireFastMutex** must be running at IRQL < DISPATCH_LEVEL. **ExAcquireFastMutex** sets the IRQL to APC_LEVEL, and the caller continues to run at APC_LEVEL after **ExAcquireFastMutex** returns. **ExAcquireFastMutex** saves the caller's previous IRQL in the mutex, however, and that IRQL is restored when the caller invokes **ExReleaseFastMutex**.

See Also

ExAcquireFastMutexUnsafe, **ExInitializeFastMutex**, **ExReleaseFastMutex**, **ExTryToAcquireFastMutex**

ExAcquireFastMutexUnsafe

```
VOID
  ExAcquireFastMutexUnsafe(
  IN PFAST_MUTEX  FastMutex
  );
```

The **ExAcquireFastMutexUnsafe** support routine acquires the given fast mutex for the current thread.

Parameters

FastMutex

Pointer to an initialized fast mutex for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Comments

ExAcquireFastMutexUnsafe puts the caller into a wait state if the given fast mutex cannot be acquired immediately. Otherwise, the caller is given ownership of the fast mutex and exclusive access to the resource it protects until it releases the fast mutex.

Any fast mutex acquired using **ExAcquireFastMutexUnsafe** must be released with **ExReleaseFastMutexUnsafe**.

Callers of **ExAcquireFastMutexUnsafe** must ensure that APCs are not delivered to the current thread while the fast mutex is held. This can be accomplished in two ways:

1. Callers can set the IRQL = APC_LEVEL before calling **ExAcquireFastMutexUnsafe** or
2. Callers can invoke **ExAcquireFastMutexUnsafe** from within a critical section by calling **KeEnterCriticalRegion** prior to calling **ExAcquireFastMutexUnsafe**.

If the caller chooses to invoke **ExAcquireFastMutexUnsafe** from within a critical section, then the caller must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAcquireFastMutex, ExInitializeFastMutex, ExReleaseFastMutexUnsafe, KeEnterCriticalRegion, KeLeaveCriticalRegion

ExAcquireResourceExclusive

```
BOOLEAN
  ExAcquireResourceExclusive(
  IN PERESOURCE  Resource,
  IN BOOLEAN  Wait
  );
```

The **ExAcquireResourceExclusive** support routine is exported to support existing driver binaries and is obsolete. Use **ExAcquireResourceExclusiveLite** instead.

ExAcquireResourceExclusiveLite

```
BOOLEAN
  ExAcquireResourceExclusiveLite(
  IN PERESOURCE  Resource,
  IN BOOLEAN  Wait
  );
```

The **ExAcquireResourceExclusiveLite** support routine acquires the given resource for exclusive access by the calling thread.

Include

ntddk.h

Parameters

Resource

Pointer to the resource to acquire.

Wait

Set to TRUE if the caller should be put into a wait state until the resource can be acquired if it cannot be acquired immediately.

Return Value

ExAcquireResourceExclusiveLite returns TRUE if the resource is acquired. This routine returns FALSE if the input *Wait* is FALSE and exclusive access cannot be granted immediately.

Comments

Normal kernel APCs must be disabled before calling **ExAcquireResourceExclusiveLite**. Otherwise a bugcheck occurs. Normal kernel APCs can be disabled by calling **KeEnterCriticalRegion** or by raising the calling thread's IRQL to APC_LEVEL.

For better performance, call **ExTryToAcquireResourceExclusiveLite**, rather than calling **ExAcquireResourceExclusiveLite** with *Wait* set to FALSE.

Callers of **ExAcquireResourceExclusiveLite** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAcquireResourceSharedLite, **ExGetExclusiveWaiterCount**, **ExGetSharedWaiterCount**, **ExInitializeResourceLite**, **ExReinitializeResourceLite**, **ExIsResourceAcquiredExclusiveLite**, **ExReleaseResourceForThreadLite**, **ExTryToAcquireResourceExclusiveLite**, **KeEnterCriticalRegion**

ExAcquireResourceShared

```
BOOLEAN
ExAcquireResourceSharedLite(
    IN PERESOURCE Resource,
    IN BOOLEAN Wait
    );
```

The **ExAcquireResourceShared** support routine is exported to support existing driver binaries, and is obsolete. Use **ExAcquireResourceSharedLite** instead.

ExAcquireResourceSharedLite

```
BOOLEAN
  ExAcquireResourceSharedLite(
  IN PERESOURCE  Resource,
  IN BOOLEAN  Wait
  );
```

The **ExAcquireResourceSharedLite** support routine acquires the given resource for shared access by the calling thread.

Parameters

Resource

Pointer to the resource to acquire.

Wait

Set to TRUE if the resource cannot be acquired immediately and if the caller should be put into a wait state until the resource can be acquired.

Include

ntddk.h

Return Value

ExAcquireResourceSharedLite returns TRUE if (or when) the resource is acquired. This routine returns FALSE if the input *Wait* is FALSE and shared access cannot be granted immediately.

Comments

Whether or when the caller is given shared access to the given resource depends on the following:

- If the resource is currently unowned, shared access is granted immediately to the current thread.
- If the caller already has acquired the resource, the current thread is granted the same type of access recursively. Note that making this call does not convert a caller's exclusive ownership of a given resource to shared.
- If the resource is currently owned as shared by another thread and no thread is waiting for exclusive access to the resource, shared access is granted to the caller immediately. The caller is put into a wait state if there is an exclusive waiter.
- If the resource is currently owned as exclusive by another thread or if there is another thread waiting for exclusive access and the caller does not already have shared access to the resource, the current thread either is put into a wait state (*Wait* set to TRUE) or **ExAcquireResourceSharedLite** returns FALSE.

Callers of **ExAcquireResourceSharedLite** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAcquireResourceExclusiveLite, ExAcquireSharedStarveExclusive, ExAcquireSharedWaitForExclusive, ExConvertExclusiveToSharedLite, ExGetExclusiveWaiterCount, ExGetSharedWaiterCount, ExInitializeResourceLite, ExReinitializeResourceLite, ExIsResourceAcquiredSharedLite, ExReleaseResourceForThreadLite

ExAcquireSharedStarveExclusive

```
BOOLEAN
  ExAcquireSharedStarveExclusive(
  IN PERESOURCE  Resource,
  IN BOOLEAN  Wait
  );
```

The **ExAcquireSharedStarveExclusive** support routine acquires a given resource for shared access without waiting for any pending attempts to acquire exclusive access to the same resource.

Parameters

Resource

Pointer to the resource to be acquired for shared access.

Wait

Set to TRUE if the caller will wait until the resource becomes available when access cannot be granted immediately.

Include

ntddk.h

Return Value

ExAcquireSharedStarveExclusive returns TRUE if the requested access is granted. This routine returns FALSE if the input *Wait* is FALSE and shared access cannot be granted immediately.

Comments

Whether or when the caller is given shared access to the given resource depends on the following:

- If the resource is currently unowned, shared access is granted immediately to the current thread.

- If the caller already has acquired the resource, the current thread is granted the same type of access recursively. Note that making this call does not convert a caller's exclusive ownership of a given resource to shared.
- If the resource is currently owned as shared by another thread, shared access is granted to the caller immediately, even if another thread is waiting for exclusive access to that resource.
- If the resource is currently owned as exclusive by another thread, the caller either is put into a wait state (*Wait* set to TRUE) or **ExAcquireSharedStarveExclusive** returns FALSE.

Callers of **ExAcquireSharedStarveExclusive** usually need quick access to a shared resource in order to save an exclusive accessor from doing redundant work. For example, a file system might call this routine to modify a cached resource, such as a BCB pinned in the cache, before the Cache Manager can acquire exclusive access to the resource and write the cache out to disk.

Callers of **ExAcquireSharedStarveExclusive** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAcquireResourceSharedLite, **ExAcquireSharedWaitForExclusive**, **ExConvertExclusiveToSharedLite**, **ExGetExclusiveWaiterCount**, **ExIsResourceAcquiredExclusiveLite**, **ExIsResourceAcquiredSharedLite**, **ExTryToAcquireResourceExclusiveLite**

ExAcquireSharedWaitForExclusive

```
BOOLEAN
  ExAcquireSharedWaitForExclusive(
  IN PERESOURCE  Resource,
  IN BOOLEAN  Wait
  );
```

The **ExAcquireSharedWaitForExclusive** support routine acquires the given resource for shared access immediately if shared access can be granted. Optionally, the caller can wait for other threads to acquire and release exclusive ownership of the resource.

Parameters

Resource

Pointer to the resource to be acquired for shared access.

Wait

Set to TRUE if the caller will wait until the resource becomes available when access cannot be granted immediately.

Include

ntddk.h

Return Value

ExAcquireSharedWaitForExclusive returns TRUE if the requested access is granted or an exclusive owner releases the resource. This routine returns FALSE if the input *Wait* is FALSE and shared access cannot be granted immediately.

Comments

Whether or when the caller is given shared access to the given resource depends on the following:

- If the resource is currently unowned, shared access is granted immediately to the current thread.
- If the caller already has exclusive access to the resource, the current thread is granted the same type of access recursively.
- If the resource is currently owned as shared and there are no pending attempts to acquire exclusive access, shared access is granted to the caller immediately.
- If the resource is currently owned as shared but there is a pending attempt to acquire exclusive access, the caller either is put into a wait state (*Wait* set to TRUE) or **ExAcquireSharedWaitForExclusive** returns FALSE.

  When the current thread waits to acquire the resource until after a pending exclusive ownership has been released, **ExAcquireSharedWaitForExclusive** returns TRUE when the current thread is granted shared access to the resource and resumes execution.

Callers of **ExAcquireSharedWaitForExclusive** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAcquireResourceSharedLite, **ExAcquireSharedStarveExclusive**, **ExConvertExclusiveToSharedLite**, **ExGetExclusiveWaiterCount**, **ExIsResourceAcquiredExclusiveLite**, **ExIsResourceAcquiredSharedLite**, **ExTryToAcquireResourceExclusiveLite**

ExAllocateFromNPagedLookasideList

```
PVOID
  ExAllocateFromNPagedLookasideList(
  IN PNPAGED_LOOKASIDE_LIST  Lookaside
  );
```

The **ExAllocateFromNPagedLookasideList** support routine returns a pointer to a nonpaged entry from the given lookaside list, or it returns a pointer to a newly allocated nonpaged entry.

Parameters

Lookaside

Pointer to the head of the lookaside list, which the caller already initialized with **ExInitializeNPagedLookasideList**.

Include

wdm.h or *ntddk.h*

Return Value

ExAllocateFromNPagedLookasideList returns a pointer to an entry if one can be allocated. Otherwise, it returns NULL.

Comments

If the given lookaside list is not empty, **ExAllocateFromNPagedLookasideList** removes the first entry from the list and returns a pointer to this entry. Otherwise, **ExAllocateFromNPagedLookasideList** either calls the *Allocate* routine specified at list initialization or **ExAllocatePoolWithTag** to return an entry pointer.

The caller then can set up the returned entry with any caller-determined data. For example, a driver might use each such fixed-size entry to set up command blocks, like SCSI SRBs, to peripheral devices on a particular type of I/O bus. The caller should release each entry with **ExFreeToNPagedLookasideList** when it is no longer in use.

Callers of **ExAllocateFromNPagedLookasideList** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExInitializeNPagedLookasideList, **ExAllocateFromPagedLookasideList**, **ExFreeToNPagedLookasideList**

ExAllocateFromPagedLookasideList

```
PVOID
  ExAllocateFromPagedLookasideList(
  IN PPAGED_LOOKASIDE_LIST  Lookaside
  );
```

The **ExAllocateFromPagedLookasideList** support routine returns a pointer to a paged entry from the given lookaside list, or it returns a pointer to a newly allocated paged entry.

Parameters

Lookaside

Pointer to the resident head of the lookaside list, which the caller already initialized with **ExInitializePagedLookasideList**.

Include

wdm.h or *ntddk.h*

Return Value

ExAllocateFromPagedLookasideList returns a pointer to an entry if one can be allocated. Otherwise, it returns NULL.

Comments

If the given lookaside list is not empty, **ExAllocateFromPagedLookasideList** removes the first entry from the list and returns a pointer to this entry. Otherwise, **ExAllocateFromPagedLookasideList** either calls the *Allocate* routine specified at list initialization or **ExAllocatePoolWithTag** to return an entry pointer.

The caller then can set up the returned entry with any caller-determined data. Because the entries in a paged lookaside list are allocated from pageable memory, access to these entries must not cause a page fault. Consequently, the user of a paged lookaside list must ensure that each such entry cannot be accessed from an arbitrary thread context or at raised IRQL. The caller should release each entry with **ExFreeToNPagedLookasideList** when it is no longer in use.

Callers of **ExAllocateFromPagedLookasideList** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExInitializePagedLookasideList, **ExFreeToPagedLookasideList**

ExAllocateFromZone

```
PVOID
  ExAllocateFromZone(
  IN PZONE_HEADER  Zone
  );
```

The **ExAllocateFromZone** support routine is exported to support existing driver binaries. This routine is obsolete; use lookaside lists instead. See *Buffer Management* in Chapter 1 for more information.

ExAllocatePool

```
PVOID
  ExAllocatePool(
  IN POOL_TYPE  PoolType,
  IN SIZE_T  NumberOfBytes
  );
```

The **ExAllocatePool** support routine allocates pool memory of the specified type and returns a pointer to the allocated block. This routine is used for general pool allocation of memory.

Parameters

PoolType

Specifies the type of pool memory to allocate. See *POOL_TYPE* for a description of the available pool memory types.

NumberOfBytes

Specifies the number of bytes to allocate.

Include

wdm.h or *ntddk.h*

Return Value

If the *PoolType* is one of the *Xxx***MustS(ucceed)** values, this call succeeds if the system has *any* available must-succeed memory, and **ExAllocatePool** returns a pointer to allocated pool memory.

ExAllocatePool returns a NULL pointer if the *PoolType* is not one of the *Xxx***MustS(ucceed)** values and not enough free pool exists to satisfy the request.

Comments

If the *NumberOfBytes* requested is >= PAGE_SIZE, a page-aligned buffer is allocated. Memory requests for <= PAGE_SIZE do not cross page boundaries. Memory requests for < PAGE_SIZE are not necessarily page-aligned but are aligned on an 8-byte boundary.

For the *PoolType* **NonPagedPoolMustSucceed**, somewhat less than PAGE_SIZE memory is available. If such a call fails to allocate sufficient memory, **ExAllocatePool** causes a system crash. Consequently, a caller should request this type of memory only if that caller needs it to prevent the system from crashing or being corrupted. Very few drivers ever encounter a situation that requires them to allocate this type of memory.

A successful allocation requesting *NumberOfBytes* < PAGE_SIZE of nonpaged pool gives the caller exactly the number of requested bytes of memory. Any successful allocation that requests *NumberOfBytes* > PAGE_SIZE wastes all unused bytes on the last-allocated page.

Callers of **ExAllocatePool** must be running at IRQL <= DISPATCH_LEVEL. A caller at DISPATCH_LEVEL must specify a **NonPaged***Xxx PoolType*. Otherwise, the caller must be running at IRQL < DISPATCH_LEVEL.

If **ExAllocatePool** returns NULL, the caller should return the NTSTATUS value STATUS_INSUFFICIENT_RESOURCES or should delay processing to another point in time.

See Also

ExAllocatePoolWithTag, **ExFreePool**

ExAllocatePoolWithQuota

```
PVOID
  ExAllocatePoolWithQuota(
  IN POOL_TYPE  PoolType,
  IN SIZE_T  NumberOfBytes
  );
```

The **ExAllocatePoolWithQuota** support routine allocates pool memory, charging quota against the current thread.

Parameters

PoolType

Specifies the type of pool memory to allocate. See *POOL_TYPE* for a description of the available pool memory types.

NumberOfBytes

Specifies the number of bytes to allocate.

Include

wdm.h or *ntddk.h*

Return Value

ExAllocatePoolWithQuota returns a pointer to the allocated pool.

If the request cannot be satisfied, **ExAllocatePoolWithQuota** raises an exception.

Comments

This routine is called by highest-level drivers that allocate memory to satisfy a request in the context of the thread that originally made the I/O request. Lower-level drivers call **ExAllocatePool** instead.

If the *NumberOfBytes* requested is >= PAGE_SIZE, a page-aligned buffer is allocated. Quota is *not* charged to the thread for allocations >= PAGE_SIZE.

Memory requests for < PAGE_SIZE are allocated within a page and do not cross page boundaries. Memory requests for < PAGE_SIZE are not necessarily page-aligned but are aligned on an 8-byte boundary.

ExAllocatePoolWithQuota raises an exception if the pool allocation fails.

Callers of **ExAllocatePoolWithQuota** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAllocatePool, ExAllocatePoolWithQuotaTag, ExFreePool

ExAllocatePoolWithQuotaTag

```
PVOID
  ExAllocatePoolWithQuotaTag(
  IN POOL_TYPE  PoolType,
  IN SIZE_T  NumberOfBytes,
  IN ULONG  Tag
  );
```

The **ExAllocatePoolWithQuotaTag** support routine allocates pool memory, charging the quota against the current thread. A call to this routine is equivalent to calling **ExAllocatePoolWithQuota,** except it inserts a caller-supplied tag before the allocation. This tag appears in any crash dump of the system that occurs.

Parameters

PoolType

Specifies the type of pool memory to allocate. See *POOL_TYPE* for a description of the available pool memory types.

NumberOfBytes

Specifies the number of bytes to allocate.

Tag

Specifies a string, delimited by single quote marks, with up to four characters. The string is usually specified in reversed order.

Include

wdm.h or *ntddk.h*

Return Value

ExAllocatePoolWithQuotaTag returns a pointer to the allocated pool.

If the request cannot be satisfied, **ExAllocatePoolWithQuotaTag** raises an exception.

Comments

During driver development on a checked build of the system, this routine can be useful for crash debugging. Calling this routine, rather than **ExAllocatePoolWithQuota**, causes the caller-supplied tag to be inserted into a crash dump of pool memory.

The *Tag* passed to this routine is more readable if its bytes are reversed when this routine is called. For example, if a caller passes 'Fred' as a *Tag*, it would appear as 'derF' if the pool is dumped or when tracking pool usage in the debugger.

Callers of **ExAllocatePoolWithQuotaTag,** like callers of **ExAllocatePoolWithQuota,** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAllocatePoolWithQuota, **ExFreePool**

ExAllocatePoolWithTag

```
PVOID
  ExAllocatePoolWithTag(
  IN POOL_TYPE  PoolType,
  IN SIZE_T  NumberOfBytes,
  IN ULONG  Tag
);
```

The **ExAllocatePoolWithTag** support routine allocates pool memory. A call to this routine is equivalent to calling **ExAllocatePool**, except that **ExAllocatePoolWithTag** inserts a caller-supplied tag before the allocation. This tag appears in any crash dump of the system that occurs.

Parameters

PoolType

Specifies the type of pool memory to allocate. See *POOL_TYPE* for a description of the available pool memory types.

NumberOfBytes

Specifies the number of bytes to allocate.

Tag

Specifies a string, delimited by single quote marks, with up to four characters. The string is usually specified in reversed order.

Include

wdm.h or *ntddk.h*

Return Value

If the *PoolType* is one of the *Xxx***MustS(ucceed)** values, and if the system has *any* available memory, this call succeeds and **ExAllocatePoolWithTag** returns a pointer to allocated pool memory.

ExAllocatePoolWithTag returns a NULL pointer if the *PoolType* is not one of the *Xxx***MustS(ucceed)** values and not enough free pool exists to satisfy the request.

Comments

During driver development on a checked build of the system, this routine can be useful for crash debugging. Calling this routine, rather than **ExAllocatePool**, inserts the caller-supplied tag into a crash dump of pool memory.

The *Tag* passed to this routine is more readable if its bytes are reversed when this routine is called. For example, if a caller passes 'Fred' as a *Tag*, it would appear as 'derF' if pool is dumped or when tracking pool usage in the debugger.

Callers of **ExAllocatePoolWithTag,** like callers of **ExAllocatePool,** can be running at IRQL DISPATCH_LEVEL only if the requested *PoolType* is one of the **NonPaged***Xxx*. Otherwise, callers must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAllocatePool, ExAllocatePoolWithQuotaTag, ExAllocatePoolWithTagPriority, ExFreePool

ExAllocatePoolWithTagPriority

```
NTKERNELAPI
PVOID
NTAPI
ExAllocatePoolWithTagPriority(
    IN POOL_TYPE PoolType,
    IN SIZE_T NumberOfBytes,
    IN ULONG Tag,
    IN EX_POOL_PRIORITY Priority
    );
```

ExAllocatePoolWithTagPriority allocates pool memory of the specified type.

Parameters

PoolType

Specifies the type of pool memory to allocate. See *POOL_TYPE* for a description of the available pool memory types.

NumberOfBytes

Specifies the number of bytes to allocate.

Tag

Specifies the four-character tag used to mark the allocated buffer. See ***ExAllocatePoolWithTag*** for a description of how to use tags.

Priority

Indicates the importance of this request.

Priority Value	Description
LowPoolPriority	Specifies the system may fail the request when it runs low on resources. Driver allocations that can recover from an allocation failure use this priority.
NormalPoolPriority	Specifies the system may fail the request when it runs very low on resources. Most drivers should use this value.
HighPoolPriority	Specifies the system must not fail the request, unless it is completely out of resources. Drivers only use this value when it is critically important for the request to succeed.

The XxxSpecialPoolOverrun and XxxSpecialPoolUnderrun variants specify how memory should be allocated when Driver Verifier (or special pool) is enabled. If the driver specifies XxxSpecialPoolUnderrun, when the Memory Manager allocates memory from special pool, it allocates it at the beginning of a physical page. If the driver specifies XxxSpecialPool-Overrun, the Memory Manager allocates it at the end of a physical page.

Include

ntddk.h

Comments

Callers of **ExAllocatePoolWithTagPriority,** like callers of **ExAllocatePoolWithTag,** can be running at IRQL DISPATCH_LEVEL only if the requested *PoolType* is one of the **NonPaged***Xxx*. Otherwise, callers must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAllocatePool, **ExAllocatePoolWithTag**, **ExAllocatePoolWithQuotaTag**, **ExFreePool**

ExConvertExclusiveToShared

```
VOID
ExConvertExclusiveToShared(
    IN PERESOURCE Resource
    );
```

The **ExConvertExclusiveToShared** support routine is exported to support existing driver binaries, and is obsolete. Use **ExConvertExclusiveToSharedLite** instead.

ExConvertExclusiveToSharedLite

```
VOID
  ExConvertExclusiveToSharedLite(
  IN PERESOURCE  Resource
  );
```

The **ExConvertExclusiveToSharedLite** support routine converts a given resource from acquired for exclusive access to acquired for shared access.

Parameters

Resource

Pointer to the resource for which the access should be converted.

Include

ntddk.h

Comments

The caller must have exclusive access to the given resource. During this conversion, the current thread and any other threads waiting for shared access to the resource are given shared access.

Callers of **ExConvertExclusiveToSharedLite** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExIsResourceAcquiredExclusiveLite

ExCreateCallback

```
NTSTATUS
  ExCreateCallback(
  OUT PCALLBACK_OBJECT  *CallbackObject,
  IN POBJECT_ATTRIBUTES  ObjectAttributes,
  IN BOOLEAN  Create,
  IN BOOLEAN  AllowMultipleCallbacks
  );
```

The **ExCreateCallback** support routine either creates a new callback object or opens an existing callback object on behalf of the caller.

Parameters

CallbackObject

Pointer to the newly created or opened callback object if the routine completes with STATUS_SUCCESS.

ObjectAttributes

Pointer to a structure that contains the callback object's attributes, previously initialized by **InitializeObjectAttributes**.

Create

Requests that a callback object be created if the requested object cannot be opened.

AllowMultipleCallbacks

Specifies whether a newly created callback object should allow multiple registered callback routines. This parameter is ignored when *Create* is FALSE or when opening an existing object.

Include

wdm.h or *ntddk.h*

Return Value

ExCreateCallback returns STATUS_SUCCESS if a callback object was opened or created. Otherwise, it returns an NTSTATUS error code to indicate the nature of the failure.

Comments

A driver calls **ExCreateCallback** to create a new callback object or to open an existing callback object. After the object has been created or opened, other components can call **ExRegisterCallback** to register callback routines with the callback object.

Before calling **ExCreateCallback**, the driver must call **InitializeObjectAttributes** to initialize the OBJECT_ATTRIBUTES structure for the callback object. The caller must specify a name for the object; otherwise, the call fails with STATUS_UNSUCCESSFUL. Unnamed callback objects are not permitted. The caller should also specify any appropriate attributes, such as OBJ_CASE_INSENSITIVE.

When all operations have been completed with the callback object, the driver must call **ObDereferenceObject** to ensure that the object is deleted to prevent a memory leak.

The system creates the following callback objects for driver use:

Callback Object Name	Usage
\Callback\SetSystemTime	The system calls any callback routines registered for this object whenever the system time changes.
\Callback\PowerState	The system calls any callback routines registered for this object whenever certain system power characteristics change. When a driver registers for callback notification (**ExRegisterCallback**), it can specify the changes for which it should be notified.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

ExRegisterCallback, ExNotifyCallback, InitializeObjectAttributes, ObDereferenceObject

ExDeleteNPagedLookasideList

```
VOID
  ExDeleteNPagedLookasideList(
  IN PNPAGED_LOOKASIDE_LIST  Lookaside
  );
```

The **ExDeleteNPagedLookasideList** support routine destroys a nonpaged lookaside list.

Parameters

Lookaside

Pointer to the head of the lookaside list to be deleted, which the caller originally set up with **ExInitializeNPagedLookasideList**.

Include

wdm.h or *ntddk.h*

Comments

ExDeleteNPagedLookasideList is the reciprocal of **ExInitializeNPagedLookasideList**. It frees any remaining entries in the specified lookaside list and then removes the list from the system-wide set of active lookaside lists.

The caller of **ExDeleteNPagedLookasideList** is responsible for subsequently releasing the memory that the caller provided to contain the list head.

Callers of **ExDeleteNPagedLookasideList** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExInitializeNPagedLookasideList

ExDeletePagedLookasideList

```
VOID
  ExDeletePagedLookasideList(
  IN PPAGED_LOOKASIDE_LIST  Lookaside
  );
```

The **ExDeletePagedLookasideList** support routine destroys a paged lookaside list.

Parameters

Lookaside

Pointer to the head of the lookaside list to be deleted, which the caller originally set up with **ExInitializePagedLookasideList**.

Include

wdm.h or *ntddk.h*

Comments

ExDeletePagedLookasideList is the reciprocal of **ExInitializePagedLookasideList**. It frees any remaining entries in the specified lookaside list and then removes the list from the system-wide set of active lookaside lists.

The caller of **ExDeletePagedLookasideList** is responsible for subsequently releasing the memory that the caller provided to contain the list head.

Callers of **ExDeletePagedLookasideList** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExInitializePagedLookasideList

ExDeleteResource

```
NTSTATUS
  ExDeleteResource(
  IN PERESOURCE  Resource
  );
```

The **ExDeleteResource** support routine is exported to support existing driver binaries and is obsolete. Use **ExDeleteResourceLite** instead.

ExDeleteResourceLite

```
NTSTATUS
  ExDeleteResourceLite(
  IN PERESOURCE  Resource
  );
```

The **ExDeleteResourceLite** support routine deletes a given resource from the system's resource list.

Parameters

Resource

Pointer to the caller-supplied storage for the initialized resource variable to be deleted.

Include

ntddk.h

Return Value

ExDeleteResourceLite returns STATUS_SUCCESS if the resource was deleted.

Comments

After calling **ExDeleteResourceLite**, the caller can free the memory it allocated for its resource.

Callers of **ExDeleteResourceLite** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExFreePool, **ExInitializeResourceLite**, **ExReinitializeResourceLite**

ExExtendZone

```
NTSTATUS
  ExExtendZone(
  IN PZONE_HEADER  Zone,
  IN PVOID  Segment,
  IN ULONG  SegmentSize
  );
```

The **ExExtendZone** support routine is exported to support existing driver binaries and is obsolete. Driver writer should use lookaside lists instead. See *Buffer Management* in Chapter 1 for more information.

ExFreePool

```
VOID
  ExFreePool(
  IN PVOID  P
  );
```

The **ExFreePool** support routine deallocates a block of pool memory.

Parameters

P

Specifies the address of the block of pool memory being deallocated.

Include

wdm.h or *ntddk.h*

Comments

This routine releases memory allocated by **ExAllocatePool, ExAllocatePoolWithTag, ExAllocatePoolWithQuota,** or **ExAllocatePoolWithQuotaTag**. The memory block must not be accessed after it is freed.

Callers of **ExAllocatePool** must be running at IRQL <= DISPATCH_LEVEL.

A caller at DISPATCH_LEVEL must have specified a **NonPaged***Xxx PoolType* when the memory was allocated. Otherwise, the caller must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAllocatePool, **ExAllocatePoolWithQuota**, **ExAllocatePoolWithQuotaTag**, **ExAllocatePoolWithTag**

ExFreeToNPagedLookasideList

```
VOID
  ExFreeToNPagedLookasideList(
  IN PNPAGED_LOOKASIDE_LIST  Lookaside,
  IN PVOID  Entry
  );
```

The **ExFreeToNPagedLookasideList** support routine returns a nonpaged entry to the given lookaside list or to nonpaged pool.

Parameters

Lookaside

Pointer to the head of the lookaside list, which the caller already initialized with **ExInitializeNPagedLookasideList**.

Entry

Pointer to the entry to be freed. The caller obtained this pointer from a preceding call to **ExAllocateFromNPagedLookasideList**.

Include

wdm.h or *ntddk.h*

Comments

ExFreeToNPagedLookasideList is the reciprocal of **ExAllocateFromNPagedLookasideList**. It releases a caller-allocated entry back to the caller's lookaside list or to nonpaged pool when that entry is no longer in use.

The same entry can be reallocated or another entry allocated later with a subsequent call to **ExAllocateFromNPagedLookasideList**. The user of the lookaside list can allocate and free such entries dynamically on an "as needed" basis until it calls **ExDeleteNPagedLookasideList**, which releases any outstanding entries in the list before it clears the system state for the given lookaside list and returns control.

If the specified lookaside list has not yet reached the system-determined maximum number of entries, **ExFreeToNPagedLookasideList** inserts the given entry at the front of the list. Otherwise, the buffer at *Entry* is released to nonpaged pool using the caller-supplied *Free* routine, if any, that was set up when the lookaside list was initialized or **ExFreePool**.

Callers of **ExFreeToNPagedLookasideList** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAllocateFromNPagedLookasideList, **ExDeleteNPagedLookasideList**, **ExInitializeNPagedLookasideList**

ExFreeToPagedLookasideList

```
VOID
  ExFreeToPagedLookasideList(
  IN PPAGED_LOOKASIDE_LIST  Lookaside,
  IN PVOID  Entry
  );
```

The **ExFreeToPagedLookasideList** support routine returns a pageable entry to the given lookaside list or to paged pool.

Parameters

Lookaside

Pointer to the resident head of the lookaside list, which the caller already initialized with **ExInitializePagedLookasideList**.

Entry

Pointer to the entry to be freed. The caller obtained this pointer from a preceding call to **ExAllocateFromPagedLookasideList**.

Include

wdm.h or *ntddk.h*

Comments

ExFreeToPagedLookasideList is the reciprocal of **ExAllocateFromPagedLookasideList**. It releases a caller-allocated entry back to the caller's lookaside list or to paged pool when that entry is no longer in use.

The same entry can be reallocated or another entry can be allocated later with a subsequent call to **ExAllocateFromPagedLookasideList**. The user of a lookaside list can allocate and free such entries dynamically, as needed, until it calls **ExDeletePagedLookasideList**. **ExDeletePagedLookasideList** releases any outstanding entries in the list before it clears the system state for the given lookaside list and returns control.

If the specified lookaside list has not yet reached the system-determined maximum number of entries, **ExFreeToPagedLookasideList** inserts the given entry at the front of the list. Otherwise, the buffer at *Entry* is released back to paged pool using the caller-supplied *Free* routine, if any, that was set up when the lookaside list was initialized or **ExFreePool**.

Callers of **ExFreeToPagedLookasideList** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAllocateFromPagedLookasideList, **ExDeletePagedLookasideList**, **ExInitializePagedLookasideList**

ExFreeToZone

```
PVOID
  ExFreeToZone(
  PZONE_HEADER  Zone,
  PVOID  Block
  );
```

The **ExFreeToZone** support routine is exported to support existing driver binaries and is obsolete. Driver writer should use lookaside lists instead. See *Buffer Management* in Chapter 1 for more information.

ExGetCurrentResourceThread

```
ERESOURCE_THREAD
  ExGetCurrentResourceThread(
  );
```

The **ExGetCurrentResourceThread** support routine identifies the current thread for a subsequent call to **ExReleaseResourceForThreadLite**.

Include

ntddk.h

Return Value

ExGetCurrentResourceThread returns the thread ID of the current thread.

Comments

Callers of **ExGetCurrentResourceThread** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExIsResourceAcquiredExclusiveLite, **ExIsResourceAcquiredSharedLite**, **ExReleaseResourceForThreadLite**

ExGetExclusiveWaiterCount

```
ULONG
  ExGetExclusiveWaiterCount(
  IN PERESOURCE  Resource
  );
```

The **ExGetExclusiveWaiterCount** support routine returns the number of waiters on exclusive access to a given resource.

Parameters

Resource

Pointer to the resource to be tested.

Include

ntddk.h

Return Value

ExGetExclusiveWaiterCount returns the number of threads currently waiting to acquire the given resource for exclusive access.

Comments

ExGetExclusiveWaiterCount can be called to get an estimate of how many other threads might be waiting to modify the data protected by a particular resource variable. The caller cannot assume that the returned value remains constant for any particular interval.

Callers of **ExGetExclusiveWaiterCount** can be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAcquireResourceExclusiveLite, **ExAcquireResourceSharedLite**, **ExAcquireSharedStarveExclusive**, **ExAcquireSharedWaitForExclusive**, **ExGetSharedWaiterCount**, **ExReleaseResourceForThreadLite**

ExGetPreviousMode

```
KPROCESSOR_MODE
  ExGetPreviousMode(
  VOID
  );
```

The **ExGetPreviousMode** support routine returns the previous processor mode for the current thread.

Return Value

ExGetPreviousMode returns a KPROCESSOR_MODE value, one of .**KernelMode** or **UserMode**. This value specifies the previous processor mode for the current thread.

Include

wdm.h or *ntddk.h*

Comments

If an I/O request can originate either in user mode or kernel mode and the caller passes pointers to data structures used for I/O, the driver must be able to determine whether the caller's pointers are valid in user mode or kernel mode.

If drivers are processing I/O requests using the normal IRP-based I/O dispatch method, they can determine the previous processor mode by checking the *RequestMode* parameter in the IRP header. (The *RequestMode* parameter is set by the I/O Manager.)

Alternatively, **ExGetPreviousMode** can be used to determine the previous processor mode. This routine is particularly useful in situations where a previous mode parameter is not available, for example, in a file driver that uses fast I/O.

Callers of **ExGetPreviousMode** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeGetCurrentThread

ExGetSharedWaiterCount

```
ULONG
  ExGetSharedWaiterCount(
  IN PERESOURCE  Resource
  );
```

The **ExGetSharedWaiterCount** support routine returns the number of waiters on shared access to a given resource.

Parameters

Resource

Pointer to the resource to be tested.

Include

ntddk.h

Return Value

ExGetSharedWaiterCount returns the number of threads currently waiting to acquire the given resource for shared access.

Comments

ExGetSharedWaiterCount can be called to get an estimate of how many other threads might be waiting to read the data protected by a particular resource variable. The caller cannot assume that the returned value remains constant for any particular interval.

Callers of **ExGetSharedWaiterCount** can be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAcquireResourceExclusiveLite, **ExAcquireResourceSharedLite**, **ExAcquireSharedStarveExclusive**, **ExAcquireSharedWaitForExclusive**, **ExGetExclusiveWaiterCount**, **ExReleaseResourceForThreadLite**

ExInitializeFastMutex

```
VOID
  ExInitializeFastMutex(
  IN PFAST_MUTEX  FastMutex
  );
```

The **ExInitializeFastMutex** support routine initializes a fast mutex variable, used to synchronize mutually exclusive access by a set of threads to a shared resource.

Parameters

FastMutex

Pointer to a caller-allocated FAST_MUTEX structure, which represents the fast mutex, in the nonpaged memory pool.

Include

wdm.h or *ntddk.h*

Comments

ExInitializeFastMutex must be called before any calls to other **Ex..FastMutex** routines occur.

Although the caller supplies the storage for the given fast mutex, the FAST_MUTEX structure is opaque: that is, its members are reserved for system use.

For better performance, use the **Ex..FastMutex** routines instead of the **Ke..Mutex** routines. However, a fast mutex cannot be acquired recursively, as a kernel mutex can.

Callers of **ExInitializeFastMutex** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAcquireFastMutex, ExAcquireFastMutexUnsafe, ExReleaseFastMutex, ExReleaseFastMutexUnsafe, ExTryToAcquireFastMutex, KeInitializeMutex

ExInitializeNPagedLookasideList

```
VOID
  ExInitializeNPagedLookasideList(
  IN PNPAGED_LOOKASIDE_LIST  Lookaside,
  IN PALLOCATE_FUNCTION  Allocate  OPTIONAL,
  IN PFREE_FUNCTION  Free  OPTIONAL,
  IN ULONG  Flags,
  IN SIZE_T  Size,
  IN ULONG  Tag,
  IN USHORT  Depth
  );
```

The **ExInitializeNPagedLookasideList** support routine initializes a lookaside list for nonpaged entries of the specified size.

Parameters

Lookaside

Pointer to the caller-supplied memory for the lookaside list head to be initialized. The caller must provide at least **sizeof**(NPAGED_LOOKASIDE_LIST) in nonpaged system space for this opaque list head.

Allocate

Either points to a caller-supplied routine for allocating an entry when the lookaside list is empty, or this parameter can be NULL. Such a caller-supplied routine is declared as follows:

```
PVOID
(*PALLOCATE_FUNCTION) (
    IN_POOL_TYPE PoolType,          // NonPagedPool
    IN ULONG  NumberOfBytes,        // value of Size
    IN ULONG  Tag                   // value of Tag
    );
```

If *Allocate* is NULL, subsequent calls to **ExAllocateFromNPagedLookasideList** automatically allocate entries whenever the lookaside list is empty.

Free

Either points to a caller-supplied routine for freeing an entry whenever the lookaside list is full, or this parameter can be NULL. Such a caller-supplied routine is declared as follows:

```
VOID
(*PFREE_FUNCTION) (
    PVOID  Buffer
    );
```

If *Free* is NULL, subsequent calls to **ExFreeToNPagedLookasideList** automatically release the given entry back to nonpaged pool whenever the list is full, that is, currently holding the system-determined maximum number of entries.

Flags

Reserved. Must be zero.

Size

Specifies the size in bytes for each nonpaged entry to be allocated subsequently.

Tag

Specifies the pool tag for lookaside list entries. The *Tag* is a string of four characters delimited by single quote marks (for example, 'derF'). The characters are usually specified in reverse order so they are easier to read when dumping pool or tracking pool usage with the **PoolHitTag** variable in the debugger.

Depth

Reserved. Must be zero.

Include

wdm.h or *ntddk.h*

Comments

After calling **ExInitializeNPagedLookasideList**, memory blocks of the caller-specified *Size* can be allocated from and freed to the lookaside list with calls to **ExAllocateFromNPagedLookasideList** and **ExFreeToNPagedLookasideList**, respectively. Such dynamically allocated and freed entries can be any data structure or fixed-size buffer that the caller uses while the system is running, particularly if the caller cannot predetermine how many such entries will be in use at any given moment. The layout and contents of each fixed-size entry are caller-determined.

ExInitializeNPagedLookasideList initializes the system state to track usage of the given lookaside list, as follows:

- Zero-initializes the counters to be maintained for entries
- Stores the entry points of the caller-supplied *Allocate* and *Free* routines, if any, or sets these entry points to **ExAllocatePoolWithTag** and **ExFreePool**, respectively

- Initializes a system spin lock to control allocations from and frees to the lookaside list in a multiprocessor-safe manner if necessary
- Stores the caller-supplied entry Size and list Tag
- Sets up the system-determined limits (minimum and maximum) on the number of entries to be held in the lookaside list, which can be adjusted subsequently if system-wide demand for entries is higher or lower than anticipated
- Sets up the system-determined flags, which control the type of memory from which entries will be allocated subsequently

The OS maintains a set of all lookaside lists currently in use. As demand for lookaside list entries and on available nonpaged memory varies while the system runs, the OS adjusts its limits for the number of entries to be held in each nonpaged lookaside list dynamically.

ExInitializeNPagedLookasideList sets up the opaque list head at the caller-supplied location but preallocates no memory for list entries. Subsequently, the initial entries are allocated dynamically as calls to **ExAllocateFromNPagedLookasideList** occur, and these initial entries are held in the lookaside list as reciprocal calls to **ExFreeToNPaged-LookasideList** occur. Entries collect in the given lookaside list until the system-determined maximum is reached, whereupon any additional entries are returned to nonpaged pool as they are freed. If the list becomes empty, allocate requests are satisfied by the *Allocate* routine specified at list initialization or by **ExAllocatePoolWithTag**.

It is more efficient to pass NULL pointers for *Allocate* and *Free* to **ExInitializeNPaged-LookasideList** if the user of a lookaside list does nothing more than allocate and release fixed-size entries within these caller-supplied routines. However, any component that uses a lookaside list might supply these routines to do additional caller-determined processing, such as tracking its own dynamic memory usage by maintaining state about the number of entries it allocates and frees.

If the caller of **ExInitializeNPagedLookasideList** supplies an *Allocate* routine, that routine must allocate entries for the lookaside list using the given input parameters when it calls **ExAllocatePoolWithTag**.

Callers of **ExInitializeNPagedLookasideList** can be running at IRQL <= DISPATCH_ LEVEL, but are usually running at PASSIVE_LEVEL.

See Also

ExAllocateFromNPagedLookasideList, ExAllocatePoolWithTag, ExDeleteNPaged-LookasideList, ExFreeToNPagedLookasideList, ExFreePool, ExInitializePaged-LookasideList

ExInitializePagedLookasideList

```
VOID
  ExInitializePagedLookasideList(
  IN PPAGED_LOOKASIDE_LIST  Lookaside,
  IN PALLOCATE_FUNCTION  Allocate  OPTIONAL,
  IN PFREE_FUNCTION  Free  OPTIONAL,
  IN ULONG  Flags,
  IN SIZE_T  Size,
  IN ULONG  Tag,
  IN USHORT  Depth
  );
```

The **ExInitializePagedLookasideList** support routine initializes a lookaside list for pageable entries of the specified size.

Parameters

Lookaside

Pointer to the caller-supplied memory for the lookaside list head to be initialized. The caller must provide at least **sizeof**(PAGED_LOOKASIDE_LIST) in *nonpaged* system space for this opaque list head, even though the entries in this lookaside list will be allocated from pageable memory.

Allocate

Either points to a caller-supplied routine for allocating an entry when the lookaside list is empty, or this parameter can be NULL. Such a caller-supplied routine is declared as follows:

```
PVOID
(*PALLOCATE_FUNCTION) (
    IN_POOL_TYPE PoolType,          // PagedPool
    IN ULONG  NumberOfBytes,        // value of Size
    IN ULONG  Tag                   // value of Tag
    );
```

If *Allocate* is NULL, subsequent calls to **ExAllocateFromPagedLookasideList** automatically allocate entries whenever the lookaside list is empty.

Free

Either points to a caller-supplied routine for freeing an entry whenever the lookaside list is full, or this parameter can be NULL. Such a caller-supplied routine is declared as follows:

```
VOID
(*PFREE_FUNCTION) (
    PVOID  Buffer
    );
```

If *Free* is NULL, subsequent calls to **ExFreeToPagedLookasideList** automatically release the given entry back to paged pool whenever the list is full, that is, currently holding the system-determined maximum number of entries.

Flags

Reserved. Must be zero.

Size

Specifies the size in bytes of each entry in the lookaside list.

Tag

Specifies the pool tag for lookaside list entries. The *Tag* is a string of four characters delimited by single quote marks (for example, 'derF'). The characters are usually specified in reverse order so they are easier to read when dumping pool or tracking pool usage with the **PoolHitTag** variable in the debugger.

Depth

Reserved. Must be zero.

Include

wdm.h or *ntddk.h*

Comments

After calling **ExInitializePagedLookasideList**, blocks of the caller-specified *Size* can be allocated from and freed to the lookaside list with calls to **ExAllocateFromPaged-LookasideList** and **ExFreeToPagedLookasideList**, respectively. Such dynamically allocated and freed entries can be any data structure or fixed-size buffer that the caller uses while the system is running, particularly if the caller cannot predetermine how many such entries will be in use at any given moment. The layout and contents of each fixed-size entry are caller-determined.

ExInitializePagedLookasideList initializes the system state to track usage of the given lookaside list, as follows:

- Zero-initializes the counters to be maintained for entries
- Stores the entry points of the caller-supplied Allocate and Free routines, if any, or sets these entry points to ExAllocatePoolWithTag and ExFreePool, respectively
- Initializes a system spin lock to control allocations from and frees to the lookaside list in a multiprocessor-safe manner if necessary
- Stores the caller-supplied entry *Size* and list *Tag*

- Sets up the system-determined limits (minimum and maximum) on the number of entries to be held in the lookaside list, which can be adjusted subsequently if system-wide demand for entries is higher or lower than anticipated
- Sets up the system-determined flags, which control the type of memory from which entries will be allocated subsequently

The OS maintains a set of all lookaside lists in use. As demand for lookaside list entries and on available paged memory varies while the system runs, the OS adjusts its limits for the number of entries to be held in each paged lookaside list dynamically.

ExInitializePagedLookasideList sets up the opaque list head at the caller-supplied location but preallocates no memory for list entries. Subsequently, the initial entries are allocated dynamically as calls to **ExAllocateFromPagedLookasideList** occur, and these initial entries are held in the lookaside list as reciprocal calls to **ExFreeToPagedLookasideList** occur. Entries collect in the given lookaside list until the system-determined maximum is reached, whereupon any additional entries are returned to paged pool as they are freed. If the list becomes empty, allocate requests are satisfied by the *Allocate* routine specified at list initialization or by **ExAllocatePoolWithTag**.

It is more efficient to pass NULL pointers for *Allocate* and *Free* to **ExInitializePaged-LookasideList** if the user of a lookaside list does nothing more than allocate and release fixed-size entries within these caller-supplied routines. However, any component that uses a lookaside list might supply these routines to do additional caller-determined processing, such as tracking its own dynamic memory usage by maintaining state about the number of entries it allocates and frees.

If the caller of **ExInitializePagedLookasideList** supplies an *Allocate* routine, that routine must allocate entries for the lookaside list using the given input parameters when it calls **ExAllocatePoolWithTag**.

Callers of **ExInitializePagedLookasideList** must be running at IRQL < DISPATCH_ LEVEL.

See Also

ExAllocateFromPagedLookasideList, ExAllocatePoolWithTag, ExDeletePaged-LookasideList, ExFreePool, ExFreeToPagedLookasideList, ExInitializeNpaged-LookasideList

ExInitializeResource

```
NTSTATUS
  ExInitializeResource(
  IN PERESOURCE  Resource
  );
```

The **ExInitializeResource** support routine is exported to support existing driver binaries and is obsolete. Use **ExInitializeResourceLite** instead.

ExInitializeResourceLite

```
NTSTATUS
  ExInitializeResourceLite(
  IN PERESOURCE  Resource
  );
```

The **ExInitializeResourceLite** support routine initializes a resource variable.

Parameters

Resource

Pointer to the caller-supplied storage, which must be at least **sizeof**(ERESOURCE), for the resource variable being initialized.

Include

ntddk.h

Return Value

ExInitializeResourceLite returns STATUS_SUCCESS.

Comments

The storage for ERESOURCE must not be allocated from paged pool.

The resource variable can be used for synchronization by a set of threads. Although the caller provides the storage for the resource variable, the ERESOURCE structure is opaque: that is, its members are reserved for system use.

Call **ExDeleteResourceLite** before freeing the memory for the resource.

Callers of **ExInitializeResourceLite** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAcquireResourceExclusiveLite, **ExAcquireResourceSharedLite**, **ExAcquireSharedStarveExclusive**, **ExAcquireSharedWaitForExclusive**, **ExConvertExclusiveToSharedLite**, **ExDeleteResourceLite**, **ExIsResourceAcquiredExclusiveLite**, **ExIsResourceAcquiredSharedLite**, **ExReinitializeResourceLite**, **ExReleaseResourceForThreadLite**, **ExTryToAcquireResourceExclusiveLite**

ExInitializeSListHead

```
VOID
  ExInitializeSListHead(
  IN PSLIST_HEADER  SListHead
  );
```

The **ExInitializeSListHead** support routine initializes the head of a sequenced, interlocked, singly linked list.

Parameters

SListHead

Pointer to caller-supplied memory for the list head to be initialized. The caller must provide at least **sizeof**(SLIST_HEADER) in nonpaged memory for this opaque list head.

Include

wdm.h or *ntddk.h*

Comments

ExInitializeSListHead initializes the system-maintained state for the S-List and sets the first-entry pointer to NULL. The caller must provide resident storage for and initialize a spin lock with **KeInitializeSpinLock** before inserting any caller-allocated entry into its initialized S-List.

The sequence number for an S-List is incremented each time an entry is inserted into or removed from the S-List. To determine the number of entries currently in an S-List, call **ExQueryDepthSList**.

Subsequent calls to **ExInterlockedPushEntrySList** and **ExInterlockedPopEntrySList** insert and remove caller-allocated entries into and from the S-List. All entries for an S-List must be allocated from nonpaged pool.

Drivers that retry I/O operations should use a doubly linked interlocked queue and the **ExInterlockedInsert/Remove..List** routines, rather than an S-List.

To manage a dynamically sized set of fixed-size entries, consider setting up a lookaside list with **ExInitializeNPageLookasideList** or **ExInitializePagedLookasideList**, instead of using an S-List.

Callers of **ExInitializeSListHead** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExInitializeNPagedLookasideList, ExInitializePagedLookasideList, ExInterlocked-InsertTailList, ExInterlockedPopEntrySList, ExInterlockedPushEntrySList, ExQueryDepthSList, ExQueueWorkItem, KeInitializeSpinLock

ExInitializeWorkItem

```
VOID
  ExInitializeWorkItem(
  IN PWORK_QUEUE_ITEM  Item,
  IN PWORKER_THREAD_ROUTINE  Routine,
  IN PVOID  Context
  );
```

The **ExInitializeWorkItem** support routine is exported to support existing driver binaries and is obsolete. Use **IoAllocateWorkItem** instead.

ExInitializeZone

```
NTSTATUS
  ExInitializeZone(
  IN PZONE_HEADER  Zone,
  IN ULONG  BlockSize,
  IN PVOID  InitialSegment,
  IN ULONG  InitialSegmentSize
  );
```

The **ExInitializeZone** support routine is exported to support existing driver binaries and is obsolete. Driver writers should use lookaside lists instead. See *Buffer Management* in Chapter 1 for more information.

ExInterlockedAddLargeInteger

```
LARGE_INTEGER
  ExInterlockedAddLargeInteger(
  IN PLARGE_INTEGER  Addend,
  IN LARGE_INTEGER  Increment,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedAddLargeInteger** support routine adds a large integer value to a given addend as an atomic operation.

Parameters

Addend

Pointer to a large integer to be adjusted by the *Increment* value.

Increment

Specifies a value to be added to *Addend.*

Lock

Pointer to a spin lock to be used to synchronize access to *Addend.*

Include

wdm.h or *ntddk.h*

Return Value

ExInterlockedAddLargeInteger returns the initial value of the *Addend.*

Comments

Support routines that do interlocked operations are assumed to be incapable of causing a page fault. That is, neither their code nor any of the data they touch can cause a page fault without bringing down the system. They use spin locks to achieve atomicity in SMP computers. The caller must provide resident storage for the *Lock*, which must be initialized with **KeInitializeSpinLock** before the initial call to an **ExInterlocked***Xxx*.

The *Lock* passed to **ExInterlockedAddLargeInteger** is used to assure that the add operation on *Addend* is atomic with respect to any other operations on the same value which synchronize with this same spin lock.

ExInterlockedAddLargeInteger masks interrupts. Consequently, it can be used for synchronization between an ISR and other device driver code, provided that the same *Lock* is never reused in a call to a routine that runs at IRQL DISPATCH_LEVEL.

Note that calls to **Interlocked***Xxx* are guaranteed to be atomic with respect to other **Interlocked***Xxx* calls without caller supplied spin locks.

Callers of **ExInterlockedAddLargeInteger** run at any IRQL.

See Also

ExInterlockedAddUlong, **InterlockedIncrement**, **InterlockedDecrement**, **KeInitializeSpinLock**

ExInterlockedAddLargeStatistic

```
VOID
ExInterlockedAddLargeStatistic (
    IN PLARGE_INTEGER Addend,
    IN ULONG Increment
    );
```

ExInterlockedAddLargeStatistic performs an interlocked addition of a ULONG increment value to a LARGE_INTEGER addend value.

Parameters

Addend

Pointer to a LARGE_INTEGER value that is incremented by the value of *Increment*.

Increment

Specifies a ULONG value that is added to the value that *Addend* points to.

Include

wdm.h or *ntddk.h*

Comments

Support routines that do interlocked operations must not cause a page fault. Neither their code nor any of the data they access can cause a page fault without bringing down the system.

ExInterlockedAddLargeStatistic masks interrupts, and can be safely used to synchronize an ISR with other driver code.

ExInterlockedAddLargeStatistic runs at any IRQL.

See Also

ExInterlockedAddLargeInteger, **ExInterlockedAddUlong**

ExInterlockedAddUlong

```
ULONG
  ExInterlockedAddUlong(
  IN PULONG  Addend,
  IN ULONG  Increment,
  PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedAddUlong** support routine adds an unsigned long value to a given unsigned integer as an atomic operation.

Parameters

Addend

Pointer to an unsigned long integer whose value is to be adjusted by the *Increment* value.

Increment

Is an unsigned long integer to be added.

Lock

Pointer to a spin lock to be used to synchronize access to the *Addend*.

Include

wdm.h or *ntddk.h*

Return Value

ExInterlockedAddUlong returns the original (unsummed) value of the *Addend*.

Comments

Consider using **InterlockedExchangeAdd** instead of this routine. **InterlockedExchange-Add** can be more efficient because it does not use a spin lock and it is inlined by the compiler.

Support routines that do interlocked operations are assumed to be incapable of causing a page fault. That is, neither their code nor any of the data they touch can cause a page fault without bringing down the system. They use spin locks to achieve atomicity in SMP computers. The caller must provide resident storage for the *Lock*, which must be initialized with **KeInitializeSpinLock** before the initial call to an **ExInterlocked***Xxx*.

The *Lock* passed to **ExInterlockedAddULong** is used to assure that the add operation on *Addend* is atomic with respect to any other operations on the same value which synchronize with this same spin lock.

ExInterlockedAddUlong masks interrupts. Consequently, it can be used for synchronization between an ISR and other driver code, provided that the same *Lock* is never reused in a call to a routine that runs at IRQL DISPATCH_LEVEL.

Note that calls to **Interlocked***Xxx* are guaranteed to be atomic with respect to other **Interlocked***Xxx* calls without caller supplied spin locks.

Callers of **ExInterlockedAddUlong** run at any IRQL.

See Also

ExInterlockedAddLargeInteger, InterlockedIncrement, InterlockedDecrement, KeInitializeSpinLock

ExInterlockedAllocateFromZone

```
PVOID
  ExInterlockedAllocateFromZone(
  IN PZONE_HEADER  Zone,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedAllocateFromZone** support routine is exported to support existing driver binaries and is obsolete. Driver writer should use lookaside lists instead. See *Buffer Management* in Chapter 1 for more information.

ExInterlockedCompareExchange64

```
LONGLONG
  ExInterlockedCompareExchange64(
  IN OUT PLONGLONG  Destination,
  IN PLONGLONG  Exchange,
  IN PLONGLONG  Comparand,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedCompareExchange64** support routine compares one integer variable to another and, if they are equal, sets the first variable to a caller-supplied value.

Parameters

Destination

Pointer to an integer that will be compared and possibly replaced.

Exchange

Pointer to an integer that will replace the one at *Destination* if the comparison results in equality.

Comparand

Pointer to an integer with which the value at *Destination* will be compared.

Lock

Pointer to a caller-allocated spin-lock that is used if the host system does not support an 8-byte atomic compare and exchange operation.

Include

wdm.h or *ntddk.h*

Return Value

ExInterlockedCompareExchange64 returns the value of the variable at *Destination* when the call occurred.

Comments

ExInterlockedCompareExchange64 tests and, possibly, replaces the value of a given variable. For most underlying microprocessors, this routine is implemented inline by the compiler to execute as an atomic operation. If a spin lock is used, this routine can only be safely used on nonpaged parameters.

If the *Destination* and *Comparand* are unequal, **ExInterlockedCompareExchange** simply returns the value of *Destination*.

ExInterlockedCompareExchange64 is atomic only with respect to other **(Ex)Interlocked***Xxx* calls.

Callers of **ExInterlockedCompareExchange64** can be running at any IRQL.

See Also

InterlockedCompareExchange, **InterlockedExchange**, **InterlockedExchangeAdd**

ExInterlockedDecrementLong

```
INTERLOCKED_RESULT
  ExInterlockedDecrementLong(
  IN PLONG  Addend,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedDecrementLong** support routine is exported to support existing driver binaries and is obsolete. Use **InterlockedDecrement** instead.

ExInterlockedExchangeAddLargeInteger

```
LARGE_INTEGER
  ExInterlockedExchangeAddLargeInteger(
  IN PLARGE_INTEGER Addend,
  IN LARGE_INTEGER Increment,
  IN PKSPIN_LOCK Lock
  );
```

The **ExInterlockedExchangeAddLargeInteger** support routine performs an atomic operation that increments the *Addend* value by the *Increment* value.

Parameters

Addend

Pointer to a value that is incremented by the value of *Increment.*

Increment

The increment value added to the value pointed to by *Addend.*

Lock

Pointer to a spin lock that is used to synchronize access to *Addend.* (*Lock* might not be used; see the **Comments** section).

Include

ntddk.h

Return Value

ExInterlockedExchangeAddLargeInteger returns the input value pointed to by *Addend.*

Comments

If supported by the processor, **ExInterlockedExchangeAddLargeInteger** uses a memory-locked, atomic exchange operation. If such an exchange operation is supported, the routine does not use the spin lock and is probably faster than **ExInterlockedAddLargeInteger**. If such an exchange operation is not supported, this routine uses the spin lock and is equivalent to **ExInterlockedAddLargeInteger**.

See Also

ExInterlockedAddLargeInteger, ExInterlockedAddUlong, InterlockedDecrement, InterlockedIncrement

ExInterlockedExchangeUlong

```
ULONG
  ExInterlockedExchangeUlong(
  IN PULONG  Target,
  IN ULONG  Value,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedExchangeUlong** support routine is exported to support existing driver binaries and is obsolete. Use **InterlockedExchange** instead.

ExInterlockedExtendZone

```
NTSTATUS
  ExInterlockedExtendZone(
  IN PZONE_HEADER  Zone,
  IN PVOID  Segment,
  IN ULONG  SegmentSize,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedExtendZone** support routine is exported to support existing driver binaries and is obsolete. Driver writer should use lookaside lists instead. See *Buffer Management* in Chapter 1 for more information.

ExInterlockedFlushSList

```
PSINGLE_LIST_ENTRY
ExInterlockedFlushSList (
  IN PSLIST_HEADER ListHead
  );
```

ExInterlockedFlushSList removes all entries on a sequenced, single-linked list (S-List) in a synchronized, multiprocessor-safe way.

Parameters

ListHead

Pointer to an S-List header.

Include

ntddk.h

Return Value

If there are entries on the specified S-List, **ExInterlockedFlushSList** returns a pointer to the first entry on the S-List; otherwise, it returns NULL.

Comment

ExInterlockedFlushSList sets the pointer to the first entry on the specified S-List to NULL.

See Also

ExInitializeSListHead

ExInterlockedFreeToZone

```
PVOID
  ExInterlockedFreeToZone(
  IN PZONE_HEADER  Zone,
  IN PVOID  Block,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedFreeToZone** support routine is exported to support existing driver binaries and is obsolete. Driver writer should use lookaside lists instead. See *Buffer Management* in Chapter 1 for more information.

ExInterlockedIncrementLong

```
INTERLOCKED_RESULT
  ExInterlockedIncrementLong(
  IN PLONG  Addend,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedIncrementLong** support routine is exported to support existing driver binaries and is obsolete. Use **InterlockedIncrement** instead.

ExInterlockedInsertHeadList

```
PLIST_ENTRY
  ExInterlockedInsertHeadList(
  IN PLIST_ENTRY  ListHead,
  IN PLIST_ENTRY  ListEntry,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedInsertHeadList** support routine inserts an entry at the head of a doubly linked list so that access to the list is synchronized in a multiprocessor-safe way.

Parameters

ListHead

Pointer to the head of the doubly linked list into which an entry is to be inserted.

ListEntry

Pointer to the entry to be inserted at the head of the list.

Lock

Pointer to a caller-supplied spin lock used to synchronize access to the list.

Include

wdm.h or *ntddk.h*

Return Value

ExInterlockedInsertHeadList returns a pointer to the entry that was at the head of the interlocked queue before this entry was inserted. If the queue was empty, it returns NULL.

Comments

Support routines that do interlocked operations are assumed to be incapable of causing a page fault. That is, neither their code nor any of the data they touch can cause a page fault without bringing down the system. They use spin locks to achieve atomicity in SMP computers. The caller must provide resident storage for the *Lock*, which must be initialized with **KeInitializeSpinLock** before the initial call to an **ExInterlocked***Xxx*. A caller *must not* be holding this spin lock when it calls **ExInterlockedInsertHeadList**.

The caller also must supply resident storage for the interlocked queue. The *ListHead* must be initialized with **InitializedListHead** before the initial call to an **ExInterlocked..List**.

If the caller uses only **ExInterlocked..List** routines to manipulate the list, then these routines can be called from a single IRQL that is <= DIRQL. If other driver routines access the list using any other routines, such as the noninterlocked **InsertHeadList**, then callers of **ExInterlocked..List** must be at <= DISPATCH_LEVEL.

Usually, drivers call **ExInterlockedInsertTailList** to insert an IRP into a driver-managed interlocked queue. They call **ExInterlockedInsertHeadList** only to requeue an IRP for a retry.

See Also

ExInterlockedInsertTailList, **ExInterlockedRemoveHeadList**, **InitializeListHead**, **KeInitializeSpinLock**

ExInterlockedInsertTailList

```
PLIST_ENTRY
  ExInterlockedInsertTailList(
  IN PLIST_ENTRY  ListHead,
  IN PLIST_ENTRY  ListEntry,
  IN PKSPIN_LOCK  Lock
);
```

The **ExInterlockedInsertTailList** support routine inserts an entry at the tail of a doubly linked list so access to the list is synchronized in a multiprocessor-safe way.

Parameters

ListHead

Pointer to the head of the doubly linked list into which an entry is to be inserted.

ListEntry

Pointer to the entry to be inserted at the tail of the list.

Lock

Pointer to a caller-supplied spin lock, used to synchronize access to the list.

Include

wdm.h or *ntddk.h*

Return Value

ExInterlockedInsertTailList returns a pointer to the entry that was at the tail of the interlocked queue before this entry was inserted. If the queue was empty, it returns NULL.

Comments

Support routines that do interlocked operations are assumed to be incapable of causing a page fault. That is, neither their code nor any of the data they touch can cause a page fault without bringing down the system. They use spin locks to achieve atomicity in SMP computers. The caller must provide resident storage for the *Lock*, which must be initialized with **KeInitializeSpinLock** before the initial call to an **ExInterlocked***Xxx*. A caller *must not* be holding this spin lock when it calls **ExInterlockedInsertTailList**.

The caller also must supply resident storage for the interlocked queue. The *ListHead* must be initialized with **InitializeListHead** before the initial call to an **ExInterlocked..List** routine.

If the caller uses only **ExInterlocked..List** routines to manipulate the list, then these routines can be called from a single IRQL that is <= DIRQL. If other driver routines access the list using any other routines, such as the noninterlocked **InsertHeadList**, then callers of **ExInterlocked..List** must be at <= DISPATCH_LEVEL.

See Also

ExInterlockedInsertHeadList, **InitializeListHead**, **KeInitializeSpinLock**

ExInterlockedPopEntryList

```
PSINGLE_LIST_ENTRY
  ExInterlockedPopEntryList(
  IN PSINGLE_LIST_ENTRY  ListHead,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedPopEntryList** support routine removes an entry from the front of a simple singly linked list so access to the queue is synchronized in a multiprocessor-safe manner.

Parameters

ListHead

Pointer to the head of the singly linked list from which an entry is to be removed.

Lock

Pointer to a caller-supplied spin lock.

Include

wdm.h or *ntddk.h*

Return Value

If the list has no entries, **ExInterlockedPopEntryList** returns a NULL pointer. Otherwise, it returns a pointer to the dequeued entry.

Comments

The **ExInterlocked..EntryList** routines manipulate a simple, singly linked list and use a spin lock for multiprocessor-safe synchronization. For greater efficiency, use the **ExInterlocked..EntrySList** routines that manipulate a sequenced, singly linked list (an S-List), instead of a simple singly linked list.

Drivers that retry I/O operations should use a doubly linked interlocked queue and the **ExInterlockedInsert/Remove..List** routines, rather than a singly linked queue or an S-List.

ExInterlockedPopEntryList removes the first entry from the specified singly linked list.

Support routines that do interlocked operations are assumed to be incapable of causing a page fault. That is, neither their code nor any of the data they touch can cause a page fault without bringing down the system. They use spin locks to achieve atomicity in SMP computers. The caller must provide resident storage for the *Lock*, which must be initialized with **KeInitializeSpinLock** before the initial call to an **ExInterlocked***Xxx*. A caller *must not* be holding this spin lock when it calls **ExInterlockedPush/PopEntryList**.

The caller also must provide the storage for the interlocked queue. The memory at *ListHead* should be zero-initialized before the initial call to **ExInterlockedPushEntryList**.

Any of the **Ex..Interlocked** routines can be called at DIRQL from a device driver's ISR or SynchCritSection routine(s), provided that other driver routines do *not* make calls to the **ExInterlocked***Xxx* while running at < DIRQL with the same spin lock. Otherwise, callers of **ExInterlockedPopEntryList** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExInitializeSListHead, **ExInterlockedPopEntrySList**, **ExInterlockedPushEntryList**, **KeInitializeSpinLock**

ExInterlockedPopEntrySList

```
PSINGLE_LIST_ENTRY
  ExInterlockedPopEntrySList(
  IN PSLIST_HEADER  ListHead,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedPopEntrySList** support routine removes the first entry from a sequenced, singly linked list so access to this queue is synchronized in a multiprocessor-safe manner.

Parameters

ListHead

Pointer to the head of the sequenced, singly linked list from which an entry is to be removed.

Lock

Pointer to a caller-supplied spin lock.

Include

wdm.h or *ntddk.h*

Return Value

ExInterlockedPopEntrySList returns a pointer to the first entry in the list. If the list was empty, it returns NULL.

Comments

ExInterlockedPopEntrySList removes the entry at the head of the list. Before calling this routine, the list must be initialized with **ExInitializeSListHead** and one or more caller-allocated entries should be inserted with **ExInterlockedPushEntrySList**.

Drivers that retry I/O operations should use a doubly linked interlocked queue and the **ExInterlockedInsert/Remove..List** routines, rather than an S-List.

The caller must provide resident storage for the *Lock*, which must be initialized with **KeInitializeSpinLock** before the first call to **ExInterlockedPushEntrySList**. A caller *must not* be holding this spin lock when it calls **ExInterlockedPush/PopEntrySList**.

Callers of **ExInterlockedPopEntrySList** can be running at IRQL <= DISPATCH_LEVEL.

See Also

ExInitializeSListHead, **ExInterlockedRemoveHeadList**, **ExInterlockedPushEntrySList**, **ExQueryDepthSList**, **KeInitializeSpinLock**

ExInterlockedPushEntryList

```
PSINGLE_LIST_ENTRY
  ExInterlockedPushEntryList(
  IN PSINGLE_LIST_ENTRY  ListHead,
  IN PSINGLE_LIST_ENTRY  ListEntry,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedPushEntryList** support routine inserts an entry at the head of a singly linked list so access to this queue is synchronized in a multiprocessor-safe way.

Parameters

ListHead

Pointer to the head of the singly linked list into which the specified entry is to be inserted.

ListEntry

Pointer to the entry to be inserted, which the caller allocated from nonpaged pool.

Lock

Pointer to a caller-supplied spin lock, already initialized with a call to **KeInitializeSpinLock**.

Include

wdm.h or *ntddk.h*

Return Value

ExInterlockedPushEntryList returns NULL if the list had no entries. Otherwise, it returns a pointer to the entry that is pushed (the previous list head).

Comments

The **ExInterlocked..EntryList** routines manipulate a simple, singly linked list and use a spin lock for multiprocessor-safe synchronization. For greater efficiency, use the **ExInterlocked..EntrySList** routines that manipulate a sequenced, singly linked list (an S-List), rather than a simple singly linked list.

Drivers that retry I/O operations should use a doubly linked interlocked queue and the **ExInterlockedInsert/Remove..List** routines, rather than a singly linked list.

ExInterlockedPushEntrySList inserts a caller-allocated entry at the front of the specified singly linked list.

Support routines that do interlocked operations are assumed to be incapable of causing a page fault. That is, neither their code nor any of the data they touch can cause a page fault without bringing down the system. They use spin locks to achieve atomicity in SMP computers. The caller must provide resident storage for the *Lock*, which must be initialized with **KeInitializeSpinLock** before the first call to **ExInterlockedPushEntryList**. A caller *must not* be holding this spin lock when it calls **ExInterlockedPush/PopEntryList**.

The caller also must provide resident storage for the head of the interlocked queue. The memory containing the *ListHead* should be zero-initialized before the initial call to **ExInterlockedPushEntryList**.

Any of the **Ex..Interlocked** routines can be called at DIRQL from a device driver's ISR or SynchCritSection routine(s), provided that other driver routines do *not* make calls to the **ExInterlocked***Xxx* while running at < DIRQL with the same spin lock. Otherwise, callers of **ExInterlockedPushEntryList** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExInitializeSListHead, **ExInterlockedInsertTailList**, **ExInterlockedPushEntrySList**, **ExInterlockedPopEntryList**, **KeInitializeSpinLock**

ExInterlockedPushEntrySList

```
PSINGLE_LIST_ENTRY
  ExInterlockedPushEntrySList(
  IN PSLIST_HEADER  ListHead,
  IN PSINGLE_LIST_ENTRY  ListEntry,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedPushEntrySList** support routine inserts an entry at the head of a sequenced, singly linked list so access to the queue is synchronized in a multiprocessor-safe manner.

Parameters

ListHead

Pointer to the head of the sequenced, singly linked list into which the specified entry is to be inserted. The given list head must be in nonpaged system space and initialized with **ExInitializeSListHead** before the first call to **ExInterlockedPushEntrySList**.

ListEntry

Pointer to the caller-allocated entry to be inserted.

Lock

Pointer to a caller-supplied spin lock, which must be initialized with **KeInitializeSpinLock** before the first call to **ExInterlockedPushEntrySList**.

Include

wdm.h or *ntddk.h*

Return Value

ExInterlockedPushEntrySList returns a pointer to the previous first entry in the list, if any. If the list was empty, it returns NULL.

Comments

ExInterlockedPushEntrySList inserts *ListEntry* at the head of the list. Before each call to this routine, the caller either allocates the entry to be inserted or reinserts an entry obtained from a preceding call to **ExInterlockedPopEntrySList**. All entries in a sequenced, singly linked interlocked queue must be allocated from nonpaged pool.

Drivers that retry I/O operations should use a doubly linked interlocked queue and the **ExInterlockedInsert/Remove..List** routines, rather than an S-List.

The caller must provide resident storage for the *ListHead* and *Lock*, which must be initialized before the first call to **ExInterlockedPushSList**. A caller *must not* be holding this spin lock when it calls **ExInterlockedPush/PopEntrySList** routine.

Callers of **ExInterlockedPushEntrySList** should be running at IRQL <= DISPATCH_LEVEL.

See Also

ExInitializeSListHead, ExInterlockedInsertTailList, ExInterlockedPopEntrySList, ExQueryDepthSList, KeInitializeSpinLock

ExInterlockedRemoveHeadList

```
PLIST_ENTRY
  ExInterlockedRemoveHeadList(
  IN PLIST_ENTRY  ListHead,
  IN PKSPIN_LOCK  Lock
  );
```

The **ExInterlockedRemoveHeadList** support routine removes an entry from the head of a doubly linked list so access to this queue is synchronized in a multiprocessor-safe manner.

Parameters

ListHead

Pointer to the head of the doubly linked list from which an entry is to be removed.

Lock

Pointer to a caller-supplied spin lock.

Include

wdm.h or *ntddk.h*

Return Value

If the list is empty, a NULL pointer is returned. Otherwise, a pointer to the dequeued entry is returned.

Comments

Support routines that do interlocked operations are assumed to be incapable of causing a page fault. That is, neither their code nor any of the data they touch can cause a page fault without bringing down the system. They use spin locks to achieve atomicity in SMP computers. The caller must provide resident storage for the *Lock*, which must be initialized with

KeInitializeSpinLock before the initial call to an **ExInterlocked***Xxx*. A caller *must not* be holding this spin lock when it calls **ExInterlockedRemoveHeadList**.

The caller also must supply resident storage for the interlocked queue. The *ListHead* must be initialized with **InitializeListHead** before the initial call to an **ExInterlocked..List** routine.

If the caller uses only **ExInterlocked..List** routines to manipulate the list, then these routines can be called from a single IRQL that is <= DIRQL. If other driver routines access the list using any other routines, such as the noninterlocked **InsertHeadList**, then callers of **ExInterlocked..List** must be at <= DISPATCH_LEVEL.

See Also

ExInterlockedInsertHeadList, **ExInterlockedInsertTailList**, **InitializeListHead**, **KeInitializeSpinLock**

ExIsFullZone

```
BOOLEAN
  ExIsFullZone(
  IN PZONE_HEADER  Zone
  );
```

The **ExIsFullZone** support routine is exported to support existing driver binaries and is obsolete. Driver writer should use lookaside lists instead. See *Buffer Management* in Chapter 1 for more information.

ExIsObjectInFirstZoneSegment

```
BOOLEAN
  ExIsObjectInFirstZoneSetment(
  IN PZONE_HEADER  Zone,
  IN PVOID  Object
  );
```

The **ExIsObjectInFirstZoneSegment** support routine is exported to support existing driver binaries and is obsolete. Driver writer should use lookaside lists instead. See *Buffer Management* in Chapter 1 for more information.

ExIsProcessorFeaturePresent

```
BOOLEAN
  ExIsProcessorFeaturePresent(
  IN ULONG ProcessorFeature
  );
```

The **ExIsProcessorFeaturePresent** support routine queries for the existence of a specified processor feature.

Parameters

ProcessorFeature

Specifies one of the following constant values:

PF_FLOATING_POINT_PRECISION_ERRATA

Pentium processor has divide bug.

PF_FLOATING_POINT_EMULATED

Processor does not have floating point hardware.

PF_COMPARE_EXCHANGE_DOUBLE

Processor has a CMPXCHG8B instruction (8-byte, memory-locked compare and exchange).

PF_MMX_INSTRUCTIONS_AVAILABLE

Processor supports MMX instructions in hardware.

PF_3DNOW_INSTRUCTIONS_AVAILABLE

Processor supports AMD 3DNow instructions.

PF_RDTSC_INSTRUCTION_AVAILABLE

Processor supports a RDTSC instruction (read timestamp counter instruction).

Include

ntddk.h

Return Value

ExIsProcessorFeaturePresent returns TRUE if the specified processor feature is present; otherwise it returns FALSE.

Comments

Callers of **ExIsProcessorFeaturePresent** must be running at IRQL PASSIVE_LEVEL.

ExIsResourceAcquiredExclusive

```
BOOLEAN
  ExIsResourceAcquiredExclusive(
  IN PERESOURCE  Resource
  );
```

The **ExIsResourceAcquiredExclusive** support routine is exported to support existing driver binaries, and is obsolete. Use **ExIsResourceAcquiredExclusiveLite** instead.

ExIsResourceAcquiredExclusiveLite

```
BOOLEAN
  ExIsResourceAcquiredExclusiveLite(
  IN PERESOURCE  Resource
  );
```

The **ExIsResourceAcquiredExclusiveLite** support routine returns whether the current thread has exclusive access to a given resource.

Parameters

Resource

Pointer to the resource to be queried.

Include

ntddk.h

Return Value

ExIsResourceAcquiredExclusiveLite returns TRUE if the caller already has exclusive access to the given resource.

Comments

Callers of **ExIsResourceAcquiredExclusiveLite** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAcquireResourceExclusiveLite, **ExIsResourceAcquiredSharedLite**, **ExTryToAcquireResourceExclusiveLite**

ExIsResourceAcquiredSharedLite

```
USHORT
  ExIsResourceAcquiredSharedLite(
  IN PERESOURCE  Resource
  );
```

The **ExIsResourceAcquiredSharedLite** support routine returns whether the current thread has shared access to a given resource.

Parameters

Resource

Pointer to the resource to be queried.

Include

ntddk.h

Return Value

ExIsResourceAcquiredSharedLite returns the number of times the caller has acquired shared access to the given resource.

Comments

Callers of **ExIsResourceAcquiredSharedLite** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAcquireResourceSharedLite, **ExAcquireSharedStarveExclusive**, **ExAcquireSharedWaitForExclusive**, **ExIsResourceAcquiredExclusiveLite**

ExLocalTimeToSystemTime

```
VOID
  ExLocalTimeToSystemTime(
  IN PLARGE_INTEGER  LocalTime,
  OUT PLARGE_INTEGER  SystemTime
  );
```

The **ExLocalTimeToSystemTime** support routine converts a system time value for the current time zone to an unbiased, GMT value.

Parameters

LocalTime

Pointer to a variable set to the locale-specific time.

SystemTime

Pointer to the returned value for GMT system time.

Include

ntddk.h

Comments

ExLocalTimeToSystemTime adds the time-zone bias at the current locale to compute the corresponding GMT system time value.

Callers of **ExLocalTimeToSystemTime** can be running at any IRQL.

See Also

ExSystemTimeToLocalTime

ExNotifyCallback

```
VOID
  ExNotifyCallback(
  IN PCALLBACK_OBJECT  CallbackObject,
  IN PVOID  Argument1,
  IN PVOID  Argument2
  );
```

The **ExNotifyCallback** support routine causes all callback routines registered for the given object to be called.

Parameters

CallbackObject

Pointer to the callback object for which all registered callback routines will be called.

Argument1

Specifies the parameter that is passed as *Argument1* of the callback routine.

Argument2

Specifies the parameter that is passed as *Argument2* of the callback routine.

Include

ntddk.h

Comments

Driver writers *must not* call **ExNotifyCallback** for any of the system-defined callback objects listed in **ExCreateCallback**.

The system calls callback routines in order of their registration.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL. The system calls all registered callback routines at the caller's IRQL.

See Also

ExCreateCallback, **ExRegisterCallback**

ExQueryDepthSList

```
USHORT
  ExQueryDepthSList(
  IN PSLIST_HEADER  SListHead
  );
```

The **ExQueryDepthSList** support routine returns the number of entries currently in a given sequenced, singly linked list.

Parameters

SListHead

Pointer to the head of the sequenced, singly linked list to be queried, which the caller has already initialized with **ExInitializeSListHead**.

Include

wdm.h or *ntddk.h*

Return Value

ExQueryDepthSList returns the current number of entries in the S-List.

Comments

Callers of **ExQueryDepthSList** can be running at IRQL <= DISPATCH_LEVEL.

See Also

ExInitializeSListHead, **ExInterlockedPushEntrySList**, **ExInterlockedPopEntrySList**

ExQueueWorkItem

```
VOID
  ExQueueWorkItem(
  IN PWORK_QUEUE_ITEM  WorkItem,
  IN WORK_QUEUE_TYPE  QueueType
  );
```

The **ExQueueWorkItem** support routine is exported to support existing driver binaries and is obsolete. Use **IoQueueWorkItem** instead.

ExRaiseAccessViolation

```
VOID
  ExRaiseAccessViolation(
  VOID
  );
```

The **ExRaiseAccessViolation** support routine can be used with structured exception handling to throw a driver-determined exception for a memory access violation that occurs when a driver processes I/O requests.

Include

ntddk.h

Comments

ExRaiseAccessViolation raises an exception with the exception code set to STATUS_ACCESS_VIOLATION.

Callers of **ExRaiseAccessViolation** must be running at IRQL PASSIVE_LEVEL.

Because **ExRaiseAccessViolation** can only be used at IRQL PASSIVE_LEVEL, only high-level drivers typically use this routine—for example, file system drivers.

See Also

ExRaiseDatatypeMisalignment, **ExRaiseStatus**, **IoAllocateErrorLogEntry**, **KeBugCheckEx**

ExRaiseDatatypeMisalignment

```
VOID
  ExRaiseDatatypeMisalignment(
  VOID
  );
```

The **ExRaiseDatatypeMisalignment** support routine can used with structured exception handling to throw a driver-determined exception for a misaligned data type that occurs when a driver processes I/O requests.

Include

ntddk.h

Comments

ExRaiseDatatypeMisalignment raises an exception with the exception code set to STATUS_DATATYPE_MISALIGNMENT.

Callers of **ExRaiseDatatypeMisalignment** must be running at IRQL PASSIVE_LEVEL.

Because **ExRaiseDatatypeMisalignment** can only be used at IRQL PASSIVE_LEVEL, only high-level drivers typically use this routine — for example, file system drivers.

See Also

ExRaiseAccessViolation, ExRaiseStatus, IoAllocateErrorLogEntry, KeBugCheckEx

ExRaiseStatus

```
VOID
  ExRaiseStatus(
  IN NTSTATUS  Status
  );
```

The **ExRaiseStatus** support routine is called by drivers that supply structured exception handlers to handle particular errors that occur while they are processing I/O requests.

Parameters

Status

Is one of the system-defined STATUS_*XXX* values.

Include

wdm.h or *ntddk.h*

Comments

Highest-level drivers, particularly file systems, can call **ExRaiseStatus**.

Callers of **ExRaiseStatus** must be running at IRQL PASSIVE_LEVEL.

See Also

ExRaiseAccessViolation, ExRaiseDatatypeMisalignment, IoAllocateErrorLogEntry, KeBugCheckEx

ExRegisterCallback

```
PVOID
  ExRegisterCallback(
  IN PCALLBACK_OBJECT  CallbackObject,
  IN PCALLBACK_FUNCTION  CallbackFunction,
  IN PVOID  CallbackContext
  );
```

The **ExRegisterCallback** support routine registers a given callback routine with a given callback object.

Parameters

CallbackObject

Pointer to a callback object obtained from **ExCreateCallback**.

CallbackFunction

Pointer to a driver callback routine, which must be nonpageable. The callback routine must conform to the following prototype:

```
VOID
(*PCALLBACK_FUNCTION ) (
    IN PVOID CallbackContext,
    IN PVOID Argument1,
    IN PVOID Argument2
    );
```

The callback routine parameters are as follows:

CallbackContext

Pointer to a driver-supplied context area as specified in the *CallbackContext* parameter of **ExRegisterCallback**.

Argument1

Pointer to a parameter defined by the callback object.

Argument2

Pointer to a parameter defined by the callback object.

CallbackContext

Pointer to a caller-defined structure of data items to be passed as the context parameter of the callback routine each time it is called. Typically the context is part of the caller's device object extension.

Include

wdm.h or *ntddk.h*

Return Value

ExRegisterCallback returns a pointer to a callback registration handle that should be treated as opaque and reserved for system use. This pointer is NULL if **ExRegisterCallback** completes with an error.

Comments

A driver calls **ExRegisterCallback** to register a callback routine with a specified callback object.

If the object allows only one registered callback routine, and such a routine is already registered, **ExRegisterCallback** returns NULL.

Callers of **ExRegisterCallback** must save the returned pointer for use later in a call to **ExUnregisterCallback**. The pointer is required when removing the callback routine from the list of registered callback routines for the callback object.

The meanings of *Argument1* and *Argument2* of the registered callback routine depend on the callback object and are defined by the component that created it. The following are the parameters for the system-defined callback objects:

\Callback\SetSystemTime

Argument 1

Not used.

Argument 2

Not used.

\Callback\PowerState

Argument 1

PO_CB_AC_STATUS – Indicates that the system has changed from A/C to battery power, or vice versa.

PO_CB_SYSTEM_POWER_POLICY – Indicates that the system power policy has changed.

PO_CB_SYSTEM_STATE_LOCK - Indicates that a system power state change is imminent. Drivers in the paging path can register for this callback to receive early warning of such a change, allowing them the opportunity to lock their code in memory before the power state changes.

Argument 2

If *Argument1* is PO_CB_AC_STATUS, *Argument2* contains TRUE if the current power source is AC and FALSE otherwise.

If *Argument1* is PO_CB_SYSTEM_POWER_POLICY, *Argument2* is not used.

If *Argument1* is PO_CB_SYSTEM_STATE_LOCK, *Argument2* contains zero if the system is about to leave S0 and one if the system has just reentered S0.

Callers of this routine must be running at IRQL < DISPATCH_LEVEL.

The system calls registered callback routines at the same IRQL at which the driver that created the callback called **ExNotifyCallback**.

See Also

ExCreateCallback, ExNotifyCallback, ExUnregisterCallback

ExReinitializeResourceLite

```
VOID
  ExReinitializeResourceLite(
  IN PERESOURCE  Resource
  );
```

The **ExReinitializeResourceLite** support routine reinitializes an existing resource variable.

Parameters

Resource

Pointer to the caller supplied resource variable to be reinitialized.

Include

ntddk.h

Return Value

ExReinitializeResourceLite returns STATUS_SUCCESS.

Comments

With a single call to **ExReinitializeResource**, a driver writer can replace three calls: one to **ExDeleteResourceLite**, another to **ExAllocatePool**, and a third to **ExInitializeResourceLite**. As contention for a resource variable increases, memory is dynamically allocated and attached to the resource in order to track this contention. As an optimization, **ExReinitializeResourceLite** retains and zeroes this previously allocated memory.

The ERESOURCE structure is opaque; that is, the members are reserved for system use.

Callers of **ExReinitializeResourceLite** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAcquireResourceExclusiveLite, ExAcquireResourceSharedLite, ExInitializeResourceLite, ExDeleteResourceLite, ExAcquireSharedStarveExclusive, ExAcquireSharedWaitForExclusive, ExConvertExclusiveToSharedLite, ExIsResourceAcquiredExclusiveLite, ExIsResourceAcquiredSharedLite, ExReleaseResourceForThreadLite, ExTryToAcquireResourceExclusiveLite

ExReleaseFastMutex

```
VOID
  ExReleaseFastMutex(
  IN PFAST_MUTEX  FastMutex
  );
```

ExReleaseFastMutex releases ownership of a fast mutex that was acquired with **ExAcquireFastMutex** or **ExTryToAcquireFastMutex**.

Parameters

FastMutex

Pointer to the fast mutex to be released.

Include

wdm.h or *ntddk.h*

Comments

ExReleaseFastMutex releases ownership of the given fast mutex and re-enables the delivery of APCs to the current thread.

It is a programming error to call **ExReleaseFastMutex** with a *FastMutex* that was acquired using **ExAcquireFastMutexUnsafe**.

Callers of **ExReleaseFastMutex** must be running at IRQL = APC_LEVEL. In most cases the IRQL will already be set to APC_LEVEL before **ExReleaseFastMutex** is called. This is because **ExAcquireFastMutex** has already set the IRQL to the appropriate value automatically. However, if the caller changes the IRQL after **ExAcquireFastMutex** returns, the caller must explicitly set the IRQL to APC_LEVEL prior to calling **ExReleaseFastMutex**.

See Also

ExAcquireFastMutex, **ExInitializeFastMutex**, **ExTryToAcquireFastMutex**

ExReleaseFastMutexUnsafe

```
VOID
  ExReleaseFastMutexUnsafe(
  IN PFAST_MUTEX  FastMutex
  );
```

The **ExReleaseFastMutexUnsafe** support routine releases ownership of a fast mutex that was acquired using **ExAcquireFastMutexUnsafe**.

Parameters

FastMutex

Pointer to the fast mutex to be released.

Include

wdm.h or *ntddk.h*

Comments

It is a programming error to call **ExReleaseFastMutexUnsafe** with a *FastMutex* that was acquired using **ExAcquireFastMutex** or **ExTryToAcquireFastMutex**.

Callers of **ExReleaseFastMutexUnsafe** must be running at IRQL = APC_LEVEL unless the caller invokes both **ExAcquireFastMutexUnsafe** and **ExReleaseFastMutexUnsafe** from within a critical section, in which case the caller must be running at IRQL <= APC_LEVEL.

See Also

ExAcquireFastMutexUnsafe, **ExInitializeFastMutex**

ExReleaseResource

```
VOID
  ExReleaseResource(
  IN PERESOURCE Resource
  );
```

The **ExReleaseResource** support routine has been superseded by the **ExReleaseResourceLite** support routine. **ExReleaseResource** is exported only to support existing driver binaries. Use **ExReleaseResourceLite** instead.

ExReleaseResourceForThread

```
VOID
  ExReleaseResourceForThreadLite(
  IN PERESOURCE  Resource,
  IN ERESOURCE_THREAD  ResourceThreadId
  );
```

The **ExReleaseResourceForThread** support routine is exported to support existing driver binaries and is obsolete. Use **ExReleaseResourceForThreadLite** instead.

ExReleaseResourceForThreadLite

```
VOID
  ExReleaseResourceForThreadLite(
  IN PERESOURCE  Resource,
  IN ERESOURCE_THREAD  ResourceThreadId
  );
```

The **ExReleaseResourceForThreadLite** support routine releases the input resource of the indicated thread.

Include

ntddk.h

Parameters

Resource

Pointer to the resource to release.

ResourceThreadId

Identifies the thread that originally acquired the resource.

Comments

Callers of **ExReleaseResourceForThreadLite** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAcquireResourceExclusiveLite, **ExAcquireResourceSharedLite**, **ExAcquireSharedStarveExclusive**, **ExAcquireSharedWaitForExclusive**, **ExGetCurrentResourceThread**, **ExInitializeResourceLite**, **ExReinitializeResourceLite**, **ExTryToAcquireResourceExclusiveLite**

ExReleaseResourceLite

```
VOID
  ExReleaseResourceLite(
  IN PERESOURCE Resource,
  );
```

The **ExReleaseResourceLite** support routine releases a specified executive resource owned by the current thread.

Parameters

Resource

Pointer to an executive resource owned by the current thread.

Include

ntddk.h

Comments

Callers of **ExReleaseResourceLite** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAcquireResourceExclusiveLite, ExAcquireResourceSharedLite, ExAcquireSharedStarveExclusive, ExAcquireSharedWaitForExclusive, ExGetCurrentResourceThread, ExInitializeResourceLite, ExReinitializeResourceLite, ExReleaseResourceLite, ExTryToAcquireResourceExclusiveLite

ExSetResourceOwnerPointer

```
VOID
  ExSetResourceOwnerPointer(
  IN PERESOURCE Resource,
  IN PVOID OwnerPointer
  );
```

The **ExSetResourceOwnerPointer** support routine sets the owner thread pointer for an executive resource.

Parameters

Resource

Pointer to an executive resource owned by the current thread.

OwnerPointer

Pointer to an owner thread pointer of type ERESOURCE_THREAD (for additional requirements, see the **Comments** section).

Include

ntddk.h

Comments

ExSetResourceOwnerPointer, used in conjunction with **ExReleaseResourceForThread-Lite**, provides a means for one thread (acting as an resource manager thread) to acquire and release resources for use by another thread (acting as a resource user thread).

After calling **ExSetResourceOwnerPointer** for a specific resource, the only other routine that can be called for that resource is **ExReleaseResourceForThreadLite**.

The resource manager thread acquires ownership of the resource and passes ownership to the user thread by calling **ExSetResourceOwnerPointer.** The caller must allocate the memory for the ERESOURCE_THREAD value pointed to by *OwnerPointer* in system memory, and this memory must remain allocated until **ExReleaseResourceForThreadLite** returns. The caller must also set the two low-order bits of the ERESOURCE_THREAD value pointed to by *OwnerPointer* to one—this encoding is used internally by the resource services to distinguish between owner and thread addresses.

When the user thread is done with the resource, the resource manager thread releases the user thread's ownership of the resource by calling **ExReleaseResourceForThreadLite**. The *ResourceThreadId* input parameter is set to the value of the *OwnerPointer* parameter used in the previous call to **ExSetResourceOwnerPointer** that gave the worker thread ownership of the resource.

Callers of **ExSetResourceOwnerPointer** must be running at IRQL <= DISPATCH_ LEVEL.

See Also

ExReleaseResourceForThreadLite

ExSetTimerResolution

```
ULONG
  ExSetTimerResolution(
  IN ULONG  DesiredTime,
  IN BOOLEAN  SetResolution
  );
```

More information on **ExSetTimerResolution** will be provided in a future DDK release.

ExSystemTimeToLocalTime

```
VOID
  ExSystemTimeToLocalTime(
  IN PLARGE_INTEGER  SystemTime,
  OUT PLARGE_INTEGER  LocalTime
  );
```

The **ExSystemTimeToLocalTime** support routine converts a GMT system time value to the local system time for the current time zone.

Parameters

SystemTime

Pointer to a variable set to the unbiased, GMT system time.

LocalTime

Pointer to the returned value for the current locale.

Include

ntddk.h

Comments

ExSystemTimeToLocalTime subtracts the time-zone bias from the GMT system time value to compute the corresponding time at the current locale.

Callers of **ExSystemTimeToLocalTime** can be running at any IRQL.

See Also

ExLocalTimeToSystemTime

ExTryToAcquireFastMutex

```
BOOLEAN
  ExTryToAcquireFastMutex(
  IN PFAST_MUTEX  FastMutex
  );
```

The **ExTryToAcquireFastMutex** support routine acquires the given fast mutex, if possible, with APCs to the current thread disabled.

Parameters

FastMutex

Pointer to the fast mutex to be acquired if it is not currently owned by another thread.

Include

ntddk.h

Return Value

ExTryToAcquireFastMutex returns TRUE if the current thread is given ownership of the fast mutex.

Comments

If the given fast mutex is currently unowned, **ExTryToAcquireFastMutex** gives the caller ownership with APCs to the current thread disabled until it releases the fast mutex.

Use **ExAcquireFastMutex** if the current thread must wait on the acquisition of the given mutex before it can do useful work.

Any fast mutex acquired with **ExTryToAcquireFastMutex** or **ExAcquireFastMutex** must be released with **ExReleaseFastMutex**.

Callers of **ExTryToAcquireFastMutex** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAcquireFastMutex, ExInitializeFastMutex, ExReleaseFastMutex

ExTryToAcquireResourceExclusiveLite

```
BOOLEAN
  ExTryToAcquireResourceExclusiveLite(
  IN PERESOURCE  Resource
  );
```

The **ExTryToAcquireResourceExclusiveLite** support routine attempts to acquire the given resource for exclusive access.

Parameters

Resource

Pointer to the resource to be acquired.

Include

ntddk.h

Return Value

ExTryToAcquireResourceExclusiveLite returns TRUE if the given resource has been acquired for the caller.

Comments

Use **ExAcquireResourceExclusiveLite** if the caller must have exclusive access to the resource before it can do further useful work.

Callers of **ExTryToAcquireResourceExclusiveLite** must be running at IRQL < DISPATCH_LEVEL.

See Also

ExAcquireResourceExclusiveLite, **ExAcquireSharedWaitForExclusive**, **ExIsResourceAcquiredExclusiveLite**

ExUnregisterCallback

```
VOID
  ExUnregisterCallback(
  IN PVOID  CallbackRegistration
  );
```

The **ExUnregisterCallback** support routine removes a callback routine previously registered with a callback object from the list of routines to be called during the notification process.

Parameters

CbRegistration

Is the pointer returned by **ExRegisterCallback** to identify this registration. This value should be treated as opaque and reserved for system use.

Include

wdm.h or *ntddk.h*

Comments

Callers of this routine must be running at IRQL < DISPATCH_LEVEL.

See Also

ExCreateCallback, **ExRegisterCallback**

ExUuidCreate

```
NTSTATUS
  ExUuidCreate(
  OUT UUID *Uuid
  );
```

The **ExUuidCreate** support routine sets a new UUID (GUID) structure.

Parameters

Uuid

Pointer to a caller-allocated UUID (GUID) structure that is set to a new UUID value.

Include

ntddk.h

Return Value

ExUuidCreate returns STATUS_SUCCESS if successful; otherwise, if the system is not ready to generate a new UUID, it returns STATUS_RETRY.

Comments

A UUID and a GUID are the same data type.

The caller can iteratively attempt to obtain a new UUID value.

This routine must run at IRQL PASSIVE_LEVEL.

InterlockedCompareExchange

```
LONG
  InterlockedCompareExchange(
  IN OUT PLONG Destination,
  IN LONG Exchange,
  IN LONG Comparand
  );
```

The **InterlockedCompareExchange** support routine performs an atomic operation that compares the input value pointed to by *Destination* with the value of *Comparand*. If the two compared values are equal, **InterlockedCompareExchange** sets the output value pointed to by *Destination* to the value of *Exchange*.

Parameters

Destination

Pointer to the input value that is compared with the value of *Comparand*.

Exchange

The output value pointed to by *Destination* if the input value pointed to by *Destination* equals the value of *Comparand*.

Comparand

The value that is compared with the input value pointed to by *Destination*.

Include

wdm.h or *ntddk.h*

Return Value

InterlockedCompareExchange returns the value pointed to by *Destination*.

Comments

InterlockedCompareExchange provides a fast, atomic way to synchronize the testing and updating of a variable that is shared by multiple threads. If the input value pointed to by *Destination* equals the value of *Comparand*, the output value of *Destination* is set to the value of *Exchange*.

InterlockedCompareExchange is designed for speed and, typically, is implemented inline by a compiler. **InterlockedCompareExchange** is atomic only with respect to other **Interlocked***Xxx* calls. It does not use a spin lock and can be safely used on pageable data.

Callers of **InterlockedCompareExchange** can be running at any IRQL.

See Also

ExInterlockedCompareExchange64, **InterlockedCompareExchangePointer**, **InterlockedDecrement**, **InterlockedExchange**, **InterlockedExchangePointer**, **InterlockedIncrement**

InterlockedCompareExchangePointer

```
PVOID
  InterlockedCompareExchangePointer(
  IN OUT PVOID *Destination,
  IN PVOID Exchange,
  IN PVOID Comparand
  );
```

The **InterlockedCompareExchangePointer** support routine performs an atomic operation that compares the input pointer value pointed to by *Destination* with the pointer value *Comparand*. If the two compared pointer values are equal, **InterlockedCompareExchangePointer** sets the output pointer value pointed to by *Destination* to the pointer value of *Exchange*.

Parameters

Destination

Pointer to the input pointer value compared with the pointer value of *Comparand*.

Exchange

The output pointer value that *Destination* points to if the input pointer value of *Destination* equals the pointer value of *Comparand*.

Comparand

The pointer value compared with the input pointer value pointed to by *Destination*.

Include

wdm.h or *ntddk.h*

Return Value

InterlockedCompareExchangePointer returns the pointer value pointed to by *Destination*.

Comments

InterlockedCompareExchangePointer provides a fast, atomic way to synchronize the testing and updating of a pointer variable that is shared by multiple threads. If the input value pointed to by *Destination* equals the value of *Comparand*, the value pointed to by *Destination* is set to the value of *Exchange*.

InterlockedCompareExchangePointer is designed for speed and, typically, is implemented inline by a compiler. **InterlockedCompareExchangePointer** is atomic only with respect to other **Interlocked***Xxx* calls. It does not use a spin lock and can be safely used on pageable data.

The **InterlockedCompareExchangePointer** routine is atomic only with respect to other **Interlocked***Xxx* calls.

Callers of **InterlockedCompareExchangePointer** can be running at any IRQL.

See Also

InterlockedCompareExchange, **InterlockedExchange**, **InterlockedExchangePointer**

InterlockedDecrement

```
LONG
  InterlockedDecrement(
  IN PLONG  Addend
  );
```

The **InterlockedDecrement** support routine decrements a caller supplied variable of type LONG as an atomic operation.

Parameters

Addend

Pointer to a variable to be decremented.

Include

wdm.h or *ntddk.h*

Return Value

InterlockedDecrement returns the decremented value.

Comments

InterlockedDecrement should be used instead of **ExInterlockedDecrementLong** because it is both more efficient and faster.

InterlockedDecrement is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

InterlockedDecrement is atomic only with respect to other Interlocked*Xxx* calls.

Callers of InterlockedDecrement can be running at any IRQL.

See Also

InterlockedExchange, **InterlockedIncrement**, **ExInterlockedAddLargeInteger**, **ExInterlockedAddUlong**

InterlockedExchange

```
LONG
  InterlockedExchange(
  IN OUT PLONG  Target,
  IN LONG  Value
  );
```

The **InterlockedExchange** support routine sets an integer variable to a given value as an atomic operation.

Parameters

Target

Pointer to a variable to be set to the supplied *Value* as an atomic operation.

Value

Specifies the value to which the variable will be set.

Include

wdm.h or *ntddk.h*

Return Value

InterlockedExchange returns the value of the variable at *Target* when the call occurred.

Comments

InterlockedExchange should be used instead of **ExInterlockedExchangeUlong**, because it is both faster and more efficient.

InterlockedExchange is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

A call to **InterlockedExchange** routine is atomic only with respect to other **Interlocked***Xxx* calls.

Callers of **InterlockedExchange** can be running at any IRQL.

See Also

InterlockedIncrement, **InterlockedDecrement**, **ExInterlockedAddLargeInteger**, **ExInterlockedAddUlong**

InterlockedExchangeAdd

```
LONG
  InterlockedExchangeAdd(
  IN OUT PLONG Addend,
  IN LONG Value
  );
```

The **InterlockedExchangeAdd** support routine adds a value to a given integer as an atomic operation and returns the original value of the given integer.

Parameters

Addend

Pointer to an integer variable.

Value

Is the value to be added to *Addend*.

Include

wdm.h or *ntddk.h*

Return Value

InterlockedExchangeAdd returns the original value of the *Addend* variable when the call occurred.

Comments

InterlockedExchangeAdd should be used instead of **ExInterlockedAddUlong** because it is both faster and more efficient.

InterlockedExchangeAddis implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

InterlockedExchangeAdd is atomic only with respect to other **Interlocked***Xxx* calls.

Callers of **InterlockedExchangeAdd** can be running at any IRQL.

See Also

InterlockedIncrement, **InterlockedDecrement**, **ExInterlockedAddLargeInteger**, **ExInterlockedAddUlong**

InterlockedExchangePointer

```
PVOID
  InterlockedExchangePointer(
  IN OUT PVOID *Target,
  IN PVOID Value
  );
```

The **InterlockedExchangePointer** support routine performs an atomic operation that sets a pointer to a new value.

Parameters

Target

Pointer to a pointer set to the value of *Value*.

Value

The new value for the pointer pointed to by *Target*.

Include

wdm.h or *ntddk.h*

Return Value

InterlockedExchangePointer returns the input value pointed to by *Target*.

Comments

InterlockedExchangePointer provides a fast, atomic way to synchronize updating a pointer variable that is shared by multiple threads.

InterlockedExchangePointer is designed for speed and, typically, is implemented inline by a compiler. **InterlockedExchangePointer** is atomic only with respect to other **Interlocked***Xxx* calls. It does not use a spin lock and can be safely used on pageable data.

A call to **InterlockedExchangePointer** is atomic only with respect to other **Interlocked***Xxx* calls.

Callers of **InterlockedExchangePointer** can be running at any IRQL.

See Also

InterlockedCompareExchange, **InterlockedCompareExchangePointer**, **InterlockedExchange**

InterlockedIncrement

```
LONG
  InterlockedIncrement(
  IN PLONG  Addend
  );
```

The **InterlockedIncrement** support routine increments a caller supplied variable as an atomic operation.

Parameters

Addend

Pointer to a variable of type LONG.

Include

wdm.h or *ntddk.h*

Return Value

InterlockedIncrement returns the incremented value.

Comments

InterlockedIncrement should be used instead of **ExInterlockedIncrementLong** because it is both more efficient and faster.

InterlockedIncrement is implemented inline by the compiler when appropriate and possible. It does not require a spin lock and can therefore be safely used on pageable data.

InterlockedIncrement is atomic only with respect to other **Interlocked***Xxx* calls.

Callers of **InterlockedIncrement** can be running at any IRQL.

See Also

InterlockedDecrement, **InterlockedExchange**, **ExInterlockedAddLargeInteger**, **ExInterlockedAddUlong**

PAGED_CODE

```
VOID PAGED_CODE();
```

The **PAGED_CODE** macro ensures that the calling thread is running at an IRQL that is low enough to permit paging. If the IRQL > APC_LEVEL, **PAGED_CODE**() causes the system to ASSERT.

Include

wdm.h or *ntddk.h*

Comments

A call to this macro should be made at the beginning of every driver routine that either contains pageable code or touches pageable code.

The **PAGED_CODE** macro only checks IRQL at the point the code executes the macro. If the code subsequently raises IRQL, it will not be detected. Driver writers should use the driver verifier to detect when the IRQL is raised improperly.

PAGED_CODE only works in checked builds.

ProbeForRead

```
VOID
  ProbeForRead (
  IN CONST VOID *Address,
  IN ULONG Length,
  IN ULONG Alignment
  );
```

The **ProbeForRead** support routine probes a structure for read accessibility and ensures correct alignment of the structure. If the structure is not accessible or has incorrect alignment, then an exception is raised.

Parameters

Address

Supplies a pointer to the structure to be probed.

Length

Length of structure.

Alignment

Supplies the required alignment of the structure expressed as the number of bytes in the primitive datatype (e.g., 1 for char, 2 for short, 4 for long, and 8 for quad).

Include

wdm.h or *ntddk.h*

Comments

Kernel-mode drivers must use **ProbeForRead** to validate read access to buffers allocated in user space. It is most commonly used during METHOD_NEITHER I/O to valid the user buffer pointed to by **Irp -> UserBuffer**.

Drivers should call **ProbeForRead** inside a **try-except** block, so that any exceptions raised are handled properly, and the driver completes the IRP with an error.

Callers of **ProbeForRead** must be running at IRQL < APC_LEVEL.

See Also

ProbeForWrite

ProbeForWrite

```
VOID
  ProbeForWrite (
  IN CONST VOID *Address,
  IN ULONG Length,
  IN ULONG Alignment
  );
```

The **ProbeForWrite** support routine probes a structure for write accessibility and ensures correct alignment of the structure. If the structure is not accessible or has incorrect alignment, then an exception is raised.

Parameters

Address

Supplies a pointer to the structure to be probed.

Length

Length of structure.

Alignment

Supplies the required alignment of the structure expressed as the number of bytes in the primitive datatype (e.g., 1 for char, 2 for short, 4 for long, and 8 for quad).

Include

wdm.h or *ntddk.h*

Comments

Kernel-mode drivers must use **ProbeForWrite** to validate write access to buffers allocated in user space. It is most commonly used during METHOD_NEITHER I/O to valid the user buffer pointed to by **Irp -> UserBuffer**.

Drivers should call **ProbeForWrite** inside a **try-except** block, so that any exceptions raised are handled properly, and the driver completes the IRP with an error.

Callers of **ProbeForWrite** must be running at IRQL < APC_LEVEL.

See Also

ProbeForRead

CHAPTER 3

Hardware Abstraction Layer Routines

References for the routines and macros described in this chapter are in alphabetical order.

For an overview of the functionality of these routines and macros, see Chapter 1, *Summary of Kernel-Mode Support Routines.*

AllocateAdapterChannel

```
NTSTATUS
  AllocateAdapterChannel(
  IN PDMA_ADAPTER DmaAdapter,
  IN PDEVICE_OBJECT DeviceObject,
  IN ULONG NumberOfMapRegisters,
  IN PDRIVER_CONTROL ExecutionRoutine,
  IN PVOID Context
  );
```

AllocateAdapterChannel prepares the system for a DMA operation on behalf of the target device object. As soon as the appropriate DMA channel and/or any necessary map registers are available, **AllocateAdapterChannel** calls a driver-supplied routine to carry out an I/O operation through the system DMA controller or a busmaster adapter.

Parameters

DmaAdapter

Points to the DMA_ADAPTER structure returned by **IoGetDmaAdapter** that represents the busmaster adapter or DMA controller.

DeviceObject

Points to the device object that represents the target device for a requested DMA operation.

NumberOfMapRegisters

Specifies the number of map registers to be used in the transfer. This value is the lesser of (the number of map registers needed to satisfy the current transfer request) and (the number of available map registers returned by **IoGetDmaAdapter**).

ExecutionRoutine

Points to a driver-supplied AdapterControl routine to be called as soon the system DMA controller or busmaster adapter is available. This routine is declared as follows:

```
IO_ALLOCATION_ACTION
(*PDRIVER_CONTROL)(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID MapRegisterBase,
    IN PVOID Context
    );
```

Context

Points to the driver-determined context to be passed to the AdapterControl routine.

Include

wdm.h or *ntddk.h*

Return Value

This routine can return one of the following NTSTATUS values:

Value	Meaning
STATUS_SUCCESS	The adapter channel has been allocated.
STATUS_INSUFFICIENT_RESOURCES	The *NumberOfMapRegisters* is larger than the value returned by **IoGetDmaAdapter**.

Comments

AllocateAdapterChannel *is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

This routine reserves exclusive access to a DMA controller channel and/or map registers for the one or more DMA operations required to satisfy the current IRP's transfer request for the specified device.

If the system DMA controller or busmaster adapter is already busy or if insufficient resources are available, the driver's request is queued until the controller or adapter is free and resources are available. Otherwise, the driver-supplied AdapterControl routine is called

immediately. Only one such request can be queued for a driver object at any one time. Therefore, the driver should not call **AllocateAdapterChannel** again for another DMA operation on the same driver object until the AdapterControl routine has completed execution. In addition, a driver must not call **AllocateAdapterChannel** from within its AdapterControl routine.

If **AllocateAdapterChannel** is called from a driver's StartIo routine to process the same IRP passed in to the StartIo routine, **AllocateAdapterChannel** passes that *Irp* to the AdapterControl routine. Otherwise, the *Irp* has no meaning, and a driver should consider the *Irp* a system-reserved parameter to its AdapterControl routine.

Drivers should save the value of *MapRegisterBase* for use when calling **FreeAdapterChannel**.

The return value of the AdapterControl routine depends on whether the device is a busmaster or uses system DMA. Drivers of busmaster devices return **DeallocateObjectKeepRegisters**; drivers of slave devices return **KeepObject**.

Callers of **AllocateAdapterChannel** must be running at IRQL DISPATCH_LEVEL.

See Also

FlushAdapterBuffers, **FreeAdapterChannel**, **FreeMapRegisters**, **IoGetDmaAdapter**, **MapTransfer**, **ReadDmaCounter**, DMA_OPERATIONS

AllocateCommonBuffer

```
PVOID
  AllocateCommonBuffer(
  IN PDMA_ADAPTER DmaAdapter,
  IN ULONG Length,
  OUT PPHYSICAL_ADDRESS LogicalAddress,
  IN BOOLEAN CacheEnabled
  );
```

AllocateCommonBuffer allocates memory and maps it so that it is simultaneously accessible from both the processor and a device for DMA operations.

Parameters

DmaAdapter

Points to the DMA_ADAPTER structure returned by **IoGetDmaAdapter** that represents the busmaster adapter or DMA controller.

Length

Specifies the number of bytes of memory to allocate.

LogicalAddress

Points to a variable that receives the logical address the device can use to access the allocated buffer. Use this address rather than calling **MmGetPhysicalAddress** because the system can take into account any platform-specific memory restrictions.

CacheEnabled

Specifies whether the allocated memory can be cached.

Include

wdm.h or *ntddk.h*

Return Value

AllocateCommonBuffer returns the base virtual address of the allocated range. If the buffer cannot be allocated, it returns NULL.

Comments

AllocateCommonBuffer *is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

AllocateCommonBuffer supports DMA in which the device and the processor continuously communicate through system memory, as in a control structure for a busmaster DMA device.

AllocateCommonBuffer also supports slave devices whose drivers use a system DMA controller's autoinitialize mode.

AllocateCommonBuffer does the following:

- Allocates memory that can be reached from both the processor and the device. This memory appears contiguous to the device.
- Allocates map registers to map the buffer, if required by the system.
- Sets up a translation for the device, including loading map registers if necessary.

To use resident system memory economically, drivers should allocate as few of these buffers per device as possible. **AllocateCommonBuffer** allocates at least a page of memory, regardless of the requested *Length*. After a successful allocation requesting fewer than PAGE_SIZE bytes, the caller can access only the requested *Length*. After a successful allocation requesting more than an integral multiple of PAGE_SIZE bytes, any remaining bytes on the last allocated page are inaccessible to the caller.

If a driver needs several pages of common buffer space, but the pages need not be contiguous, the driver should make several one-page requests to **AllocateCommonBuffer** instead of one large request. This approach conserves contiguous memory.

Drivers typically call **AllocateCommonBuffer** as part of device start-up, during their response to a PnP IRP_MN_START_DEVICE request. After start-up, it is possible that only one-page requests will succeed, if any.

Callers of **AllocateCommonBuffer** must be running at IRQL PASSIVE_LEVEL.

See Also

FreeCommonBuffer, **IoGetDmaAdapter**, DMA_OPERATIONS

FlushAdapterBuffers

```
BOOLEAN
  FlushAdapterBuffers(
  IN PDMA_ADAPTER DmaAdapter,
  IN PMDL Mdl,
  IN PVOID MapRegisterBase,
  IN PVOID CurrentVa,
  IN ULONG Length,
  IN BOOLEAN WriteToDevice
  );
```

FlushAdapterBuffers flushes any data remaining in the system DMA controller's internal cache or in a busmaster adapter's internal cache at the end of a DMA transfer operation.

Parameters

DmaAdapter

Points to the DMA_ADAPTER structure returned by **IoGetDmaAdapter** that represents the busmaster adapter or DMA controller.

Mdl

Points to the MDL that describes the buffer previously passed in the driver's call to **MapTransfer**.

MapRegisterBase

Points to the handle passed to the driver's AdapterControl routine by **AllocateAdapterChannel**.

CurrentVa

Points to the current virtual address in the buffer, described by the *Mdl*, where the I/O operation occurred. This value must be the same as the initial *CurrentVa* value passed to **MapTransfer**.

Length

Specifies the length, in bytes, of the buffer.

WriteToDevice

Specifies the direction of the DMA transfer operation: TRUE for a transfer from a buffer in system memory to the driver's device.

Include

wdm.h or *ntddk.h*

Return Value

FlushAdapterBuffers returns TRUE if any data remaining in the DMA controller's or bus-master adapter's internal cache has been successfully flushed into system memory or out to the device.

Comments

FlushAdapterBuffers *is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

To ensure that a DMA transfer is complete, every driver that performs DMA operations must call FlushAdapterBuffers before completing the IRP that requested the DMA transfer and before freeing the map registers.

A driver can get the initial *CurrentVa* for the start of a packet-based DMA transfer by calling MmGetMdlVirtualAddress. However, the value returned is an index into the *Mdl*, rather than a valid virtual address. If the driver must split a large transfer request into more than one DMA operation, it must update *CurrentVa* and *Length* for each DMA operation.

Callers of FlushAdapterBuffers must be running at IRQL <= DISPATCH_LEVEL.

See Also

AllocateAdapterChannel, **IoGetDmaAdapter**, **KeFlushIoBuffers**, **MapTransfer**, **MmGetMdlVirtualAddress**, DMA_OPERATIONS

FreeAdapterChannel

```
VOID
  FreeAdapterChannel(
  IN PDMA_ADAPTER DmaAdapter
  );
```

FreeAdapterChannel releases the system DMA controller when a driver has completed all DMA operations necessary to satisfy the current IRP.

Parameters

DmaAdapter

Points to the DMA_ADAPTER structure returned by **IoGetDmaAdapter** that represents the busmaster adapter or DMA controller.

Include

wdm.h or *ntddk.h*

Comments

FreeAdapterChannel *is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

After a driver has transferred all the data and called **FlushAdapterBuffers**, it calls **FreeAdapterChannel** to release the system DMA controller that was previously allocated with a call to **AllocateAdapterChannel**.

FreeAdapterChannel frees any map registers that were allocated by an earlier call to **AllocateAdapterChannel.** A driver calls this routine only if its AdapterControl routine returns **KeepObject**.

Callers of **FreeAdapterChannel** must be running at IRQL DISPATCH_LEVEL.

See Also

AllocateAdapterChannel, **FlushAdapterBuffers**, **FreeMapRegisters**, **IoGetDmaAdapter**, **MapTransfer**, DMA_OPERATIONS

FreeCommonBuffer

```
VOID
  FreeCommonBuffer(
  IN PDMA_ADAPTER DmaAdapter,
  IN ULONG Length,
  IN PHYSICAL_ADDRESS LogicalAddress,
  IN PVOID VirtualAddress,
  IN BOOLEAN CacheEnabled
  );
```

FreeCommonBuffer frees a common buffer allocated by **AllocateCommonBuffer**, along with all resources the buffer uses.

Parameters

DmaAdapter

Points to the DMA_ADAPTER structure returned by **IoGetDmaAdapter** that represents the busmaster adapter or DMA controller.

Length

Specifies the number of bytes to deallocate.

LogicalAddress

Specifies the logical address of the allocated memory range.

VirtualAddress

Points to the corresponding virtual address of the allocated memory range.

CacheEnabled

Indicates whether the allocated memory is cached.

Include

wdm.h or *ntddk.h*

Comments

FreeCommonBuffer *is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

To release a common buffer, a driver calls **FreeCommonBuffer** to unmap both its logical and virtual addresses. The parameters passed to **FreeCommonBuffer** must match exactly those passed to and returned from **AllocateCommonBuffer**. A driver cannot free part of an allocated common buffer.

Callers of **FreeCommonBuffer** must be running at IRQL PASSIVE_LEVEL.

See Also

AllocateCommonBuffer, **IoGetDmaAdapter**, DMA_OPERATIONS

FreeMapRegisters

```
VOID
  FreeMapRegisters(
  IN PDMA_ADAPTER DmaAdapter,
  PVOID MapRegisterBase,
  ULONG NumberOfMapRegisters
  );
```

FreeMapRegisters releases a set of map registers that were saved from a call to **AllocateAdapterChannel**.

Parameters

DmaAdapter

Points to the DMA_ADAPTER structure returned by **IoGetDmaAdapter** that represents the busmaster adapter or DMA controller.

MapRegisterBase

Points to the handle returned by the driver's call to **AllocateAdapterChannel**.

NumberOfMapRegisters

Specifies the number of map registers to be released. This value must match the number specified in an earlier call to **AllocateAdapterChannel**.

Include

wdm.h or *ntddk.h*

Comments

FreeMapRegisters *is not a system routine that can be called directly by name. This routine is only callable by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

When the driver of a busmaster device has completed the current packet-based DMA transfer request, it calls **FreeMapRegisters** to release the map registers previously allocated by a call to **AllocateAdapterChannel** and retained because its AdapterControl routine returned **DeallocateObjectKeepRegisters**. The driver must call **FreeMapRegisters** after calling **FlushAdapterBuffers.**

Callers of **FreeMapRegisters** must be running at IRQL DISPATCH_LEVEL.

See Also

AllocateAdapterChannel, **IoGetDmaAdapter**, **MapTransfer**, DMA_OPERATIONS

GetDmaAlignment

```
ULONG
  GetDmaAlignment(
  IN PDMA_ADAPTER DmaAdapter
  );
```

GetDmaAlignment returns the alignment requirements of the DMA system.

Parameters

DmaAdapter

Points to the DMA_ADAPTER structure returned by **IoGetDmaAdapter** that represents the busmaster adapter or DMA controller.

Include

wdm.h or *ntddk.h*

Return Value

GetDmaAlignment returns the alignment requirements of the DMA system.

Comments

GetDmaAlignment *is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

A driver can call this routine to determine alignment requirements for DMA buffers it allocates. The returned value should be used to set the **AlignmentRequirement** field in the device object. A driver may need to increase this value because of additional hardware device restrictions.

Callers of **GetDmaAlignment** must be running at IRQL PASSIVE_LEVEL.

See Also

IoGetDmaAdapter, DMA_OPERATIONS , DEVICE_OBJECT

GetScatterGatherList

```
NTSTATUS
GetScatterGatherList (
  IN PDMA_ADAPTER DmaAdapter,
  IN PDEVICE_OBJECT DeviceObject,
  IN PMDL Mdl,
  IN PVOID CurrentVa,
  IN ULONG Length,
  IN PDRIVER_LIST_CONTROL ExecutionRoutine,
  IN PVOID Context,
  IN BOOLEAN WriteToDevice
  );
```

GetScatterGatherList prepares the system for a DMA operation on behalf of the target device object through either the system DMA controller or a busmaster adapter. As soon as the appropriate DMA channel and any necessary map registers are available, **GetScatterGatherList** creates a scatter/gather list, initializes the map registers, and then calls a driver-supplied routine to carry out the I/O operation.

Parameters

DmaAdapter

Points to the DMA_ADAPTER structure returned by **IoGetDmaAdapter** that represents the busmaster adapter or DMA controller.

DeviceObject

Points to the device object that represents the target device for the DMA operation.

Mdl

Points to the MDL that describes the buffer at **MdlAddress** in the current IRP.

CurrentVa

Points to the current virtual address in the MDL for the buffer to be mapped for a DMA transfer operation.

Length

Specifies the length, in bytes, to be mapped.

ExecutionRoutine

Points to a driver-supplied AdapterControl routine to be called when the system DMA controller or busmaster adapter is available. This routine is declared as follows:

```
VOID
(*PDRIVER_LIST_CONTROL)(
    IN struct _DEVICE_OBJECT *DeviceObject,
    IN struct _IRP *Irp,
    IN PSCATTER_GATHER_LIST ScatterGather,
    IN PVOID Context
    );
```

Context

Points to the driver-determined context passed to the driver's Execution routine when it is called.

WriteToDevice

Indicates the direction of the DMA transfer: TRUE for a transfer from the buffer to the device, and FALSE otherwise.

Include

wdm.h or *ntddk.h*

Return Value

This routine can return one of the following NTSTATUS values:

Value	Meaning
STATUS_SUCCESS	The operation succeeded.
STATUS_INSUFFICIENT_RESOURCES	The routine could not allocate sufficient memory or the number of map registers required for the transfer is larger than the value returned by **IoGetDmaAdapter**.
STATUS_BUFFER_TOO_SMALL	The buffer is too small for the requested transfer.

Comments

GetScatterGatherList *is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

GetScatterGatherList combines the actions of the **AllocateAdapterChannel** and **MapTransfer** routines for drivers that perform scatter/gather DMA. **GetScatterGatherList** determines how many map registers are required for the transfer, allocates the map registers, maps the buffers for DMA, and fills in the scatter/gather list. It then calls the supplied

AdapterControl routine, passing a pointer to the scatter/gather list in *ScatterGather*. The driver should retain this pointer for use when calling **PutScatterGatherList**. Note that **GetScatterGatherList** does not have the queuing restrictions that apply to **AllocateAdapterChannel**.

In its AdapterControl routine, the driver should perform the I/O. On return from the driver-supplied routine, **GetScatterGatherList** keeps the map registers but frees the DMA adapter structure. The driver must call **PutScatterGatherList** (which flushes the buffers) before it can access the data in the buffer.

This routine can handle chained MDLs, provided that the total number of map registers required by all chained MDLs does not exceed the number available.

Callers of **GetScatterGatherList** must be running at IRQL DISPATCH_LEVEL.

See Also

IoGetDmaAdapter, **PutScatterGatherList**, **AllocateAdapterChannel**, DMA_OPERATIONS, SCATTER_GATHER_LIST

HalAllocateCommonBuffer

```
PVOID
  HalAllocateCommonBuffer(
  IN PADAPTER_OBJECT  AdapterObject,
  IN ULONG  Length,
  OUT PPHYSICAL_ADDRESS  LogicalAddress,
  IN BOOLEAN  CacheEnabled
  );
```

HalAllocateCommonBuffer is obsolete and is exported only to support existing driver binaries. See ***AllocateCommonBuffer*** instead.

HalAssignSlotResources

```
NTSTATUS
  HalAssignSlotResources(
  IN PUNICODE_STRING  RegistryPath,
  IN PUNICODE_STRING  DriverClassName,
  IN PDRIVER_OBJECT  DriverObject,
  IN PDEVICE_OBJECT  DeviceObject,
  IN INTERFACE_TYPE  BusType,
  IN ULONG  BusNumber,
  IN ULONG  SlotNumber,
  IN OUT PCM_RESOURCE_LIST  *AllocatedResources
  );
```

HalAssignSlotResources is *obsolete* and is exported only to support existing drivers.

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**.

See Also

CM_RESOURCE_LIST, **ExFreePool**, **HalGetBusData**, **IoAssignResources**, **IoReportDetectedDevice**, **IoReportResourceForDetection**

HalExamineMBR

```
VOID
  HalExamineMBR(
  IN PDEVICE_OBJECT  DeviceObject,
  IN ULONG  SectorSize,
  IN ULONG  MBRTypeIdentifier,
  OUT PVOID  Buffer,
  );
```

HalExamineMBR reads the master boot record (MBR) of a disk and returns data from the MBR if the MBR is of the type specified by the caller.

Parameters

DeviceObject

Points to the device object for the device being examined.

SectorSize

Specifies the minimum number of bytes that an I/O operation can fetch from the device being examined. If this value is less than 512, **HalExamineMBR** reads 512 bytes to ensure that it reads an entire partition table.

MBRTypeIdentifier

Specifies the type of MBR that may be on the disk.

Buffer

Points to a buffer that returns data from the MBR. The layout of the buffer depends on the *MBRTypeIdentifier*. The caller must deallocate this buffer as soon as possible with **ExFreePool**. This routine returns NULL in *Buffer* if the *MBRTypeIdentifier* of the disk does not match that specified by the caller or if there is an error.

Include

ntddk.h

Comments

Callers of **HalExamineMBR** must be running at IRQL PASSIVE_LEVEL.

See Also

ExFreePool

HalFreeCommonBuffer

```
VOID
  HalFreeCommonBuffer(
  IN PADAPTER_OBJECT  AdapterObject,
  IN ULONG  Length,
  IN PHYSICAL_ADDRESS  LogicalAddress,
  IN PVOID  VirtualAddress,
  IN BOOLEAN  CacheEnabled
  );
```

HalFreeCommonBuffer is obsolete and is exported only to support existing driver binaries. See ***FreeCommonBuffer*** instead.

HalGetAdapter

```
PADAPTER_OBJECT
  HalGetAdapter(
  IN PDEVICE_DESCRIPTION  DeviceDescription,
  IN OUT PULONG  NumberOfMapRegisters
  );
```

HalGetAdapter is obsolete and is exported only for existing driver binaries. See ***IoGetDmaAdapter*** instead.

HalGetBusData

```
ULONG
  HalGetBusData(
  IN BUS_DATA_TYPE  BusDataType,
  IN ULONG  BusNumber,
  IN ULONG  SlotNumber,
  IN PVOID  Buffer,
  IN ULONG  Length
  );
```

HalGetBusData is *obsolete* and is exported only to support existing drivers.

Drivers should use the PnP Manager's IRP_MN_QUERY_INTERFACE and IRP_MN_READ_CONFIG requests instead.

See Also

CM_EISA_FUNCTION_INFORMATION, CM_EISA_SLOT_INFORMATION, CM_MCA_POS_DATA, **HalAssignSlotResources**, **HalGetAdapter**, **HalGetBusDataByOffset**, **HalGetInterruptVector**, **HalSetBusData**, **HalTranslateBusAddress**, **IoAssignResources**, PCI_COMMON_CONFIG, PCI_SLOT_NUMBER, IRP_MN_QUERY_INTERFACE, IRP_MN_READ_CONFIG

HalGetBusDataByOffset

```
ULONG
  HalGetBusDataByOffset(
  IN BUS_DATA_TYPE  BusDataType,
  IN ULONG  BusNumber,
  IN ULONG  SlotNumber,
  IN PVOID  Buffer,
  IN ULONG  Offset,
  IN ULONG  Length
  );
```

HalGetBusData is *obsolete* and is exported only to support existing drivers.

Drivers should use the PnP Manager's IRP_MN_QUERY_INTERFACE request instead.

See Also

HalAssignSlotResources, **HalGetBusData**, **HalSetBusDataByOffset**, **HalTranslateBusAddress**, **IoAssignResources**, PCI_COMMON_CONFIG, PCI_SLOT_NUMBER, IRP_MN_QUERY_INTERFACE, IRP_MN_READ_CONFIG

HalGetDmaAlignmentRequirement

```
ULONG
  HalGetDmaAlignmentRequirement(
  );
```

HalGetDmaAlignmentRequirement is obsolete and exported only to support existing drivers. See ***GetDmaAlignment*** instead.

HalGetInterruptVector

```
ULONG
  HalGetInterruptVector(
  IN INTERFACE_TYPE  InterfaceType,
  IN ULONG  BusNumber,
  IN ULONG  BusInterruptLevel,
  IN ULONG  BusInterruptVector,
```

```
OUT PKIRQL  Irql,
OUT PKAFFINITY  Affinity
);
```

HalGetInterruptVector is *obsolete* and is exported only to support existing drivers.

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**.

HalReadDmaCounter

```
ULONG
  HalReadDmaCounter(
  IN PADAPTER_OBJECT  AdapterObject
  );
```

HalReadDmaCounter is obsolete and exported only to support existing driver binaries. See **ReadDmaCounter** instead.

HalSetBusData

```
ULONG
  HalSetBusData(
  IN BUS_DATA_TYPE  BusDataType,
  IN ULONG  BusNumber,
  IN ULONG  SlotNumber,
  IN PVOID  Buffer,
  IN ULONG  Length
  );
```

HalSetBusData is *obsolete* and is exported only to support existing drivers.

Drivers should use the PnP Manager's IRP_MN_QUERY_INTERFACE and IRP_MN_WRITE_CONFIG requests instead.

HalSetBusData sets bus-configuration data for a given slot or address on a particular bus.

Parameters

BusDataType

Specifies the type of bus data to be set. Currently, its value can be the following: **Cmos** or **PCIConfiguration**. However, additional types of bus configuration will be supported in future versions of the operating system. The upper bound on the bus types supported is always **MaximumBusDataType**.

BusNumber

Specifies the zero-based and system-assigned number of the bus in systems with more than one bus of the same *BusDataType*.

SlotNumber

Specifes the logical slot number for the device. When **PCIConfiguration** is specified, this is a PCI_SLOT_NUMBER-type value.

Buffer

Points to a caller-supplied buffer containing information specific to *BusDataType*.

When **Cmos** is specified, the buffer contains data to be written to CMOS (*BusNumber* equals zero) or ECMOS (*BusNumber* equals one) locations starting with the location specified by the *SlotNumber*.

When **PCIConfiguration** is specified, the buffer contains some or all of the PCI_COMMON_CONFIG information for the given *SlotNumber*. The specified *Length* determines how much information is supplied. Certain members of PCI_COMMON_CONFIG have read-only values, and the caller is responsible for preserving the system-supplied values of read-only members.

Length

Specifies the number of bytes of configuration data in *Buffer*.

Include

ntddk.h

Return Value

HalSetBusData returns the number of bytes of data successfully set for the given *SlotNumber*. If the given *BusDataType* is not valid for the current platform or if the supplied information is invalid, this routine returns zero.

Comments

Calling **HalSetBusDataByOffset** with a *BusDataType* of **PCIConfiguration** and an input *Offset* of zero is the same as calling **HalSetBusData**.

If the input *BusDataType* is **PCIConfiguration**, callers of **HalSetBusData** can be running at IRQL <= DISPATCH_LEVEL. Otherwise, callers of **HalSetBusData** must be running at IRQL PASSIVE_LEVEL.

See Also

HalGetBusData, **HalGetBusDataByOffset**, **HalSetBusDataByOffset**, PCI_COMMON_CONFIG, PCI_SLOT_NUMBER, IRP_MN_QUERY_INTERFACE, IRP_MN_WRITE_CONFIG

HalSetBusDataByOffset

```
ULONG
  HalSetBusDataByOffset(
  IN BUS_DATA_TYPE  BusDataType,
  IN ULONG  BusNumber,
  IN ULONG  SlotNumber,
  IN PVOID  Buffer,
  IN ULONG  Offset,
  IN ULONG  Length
  );
```

HalSetBusDataByOffset is *obsolete* and is exported only to support existing drivers.

Drivers should use the PnP Manager's IRP_MN_QUERY_INTERFACE and IRP_MN_WRITE_CONFIG requests instead.

See Also

HalAssignSlotResources, **HalGetBusData**, **HalGetBusDataByOffset**, **HalTranslateBusAddress**, **IoAssignResources**, PCI_COMMON_CONFIG, PCI_SLOT_NUMBER, IRP_MN_QUERY_INTERFACE, IRP_MN_WRITE_CONFIG

HalTranslateBusAddress

```
BOOLEAN
  HalTranslateBusAddress(
  IN INTERFACE_TYPE  InterfaceType,
  IN ULONG  BusNumber,
  IN PHYSICAL_ADDRESS  BusAddress,
  IN OUT PULONG  AddressSpace,
  OUT PPHYSICAL_ADDRESS  TranslatedAddress
  );
```

HalTranslateBusAddress is *obsolete* and is exported only to support existing drivers.

The PnP Manager passes lists of raw and translated resources in its IRP_MN_START_DEVICE request for each device. Consequently, PnP drivers seldom, if ever, need to translate bus addresses. However, if translation is required, drivers should use the PnP IRP_MN_QUERY_INTERFACE request to get the standard bus interface.

See Also

HalAssignSlotResources, **HalGetBusData**, **HalGetBusDataByOffset**, **IoAssignResources**, **MmMapIoSpace**, IRP_MN_QUERY_INTERFACE

MapTransfer

```
PHYSICAL_ADDRESS
  MapTransfer(
  IN PDMA_ADAPTER  DmaAdapter,
  IN PMDL  Mdl,
  IN PVOID  MapRegisterBase,
  IN PVOID  CurrentVa,
  IN OUT PULONG  Length,
  IN BOOLEAN  WriteToDevice
  );
```

MapTransfer sets up map registers for an adapter object to map a DMA transfer from a locked-down buffer.

Parameters

DmaAdapter

Points to the DMA adapter object returned by **IoGetDmaAdapter** and previously passed to **AllocateAdapterChannel** for the current IRP's transfer request.

Mdl

Points to one of the following: the MDL that describes the buffer at **MdlAddress** in the current IRP or the MDL that describes the common buffer set up by the driver of a slave device (auto-initialize mode).

MapRegisterBase

Points to the handle previously returned by **AllocateAdapterChannel** for the current IRP.

CurrentVa

Points to the current virtual address of the data to be transferred for a DMA transfer operation.

Length

Specifies the length, in bytes, to be mapped. If the driver indicated that its device was a busmaster with scatter/gather support when it called **IoGetDmaAdapter**, the value of *Length* on return from **MapTransfer** indicates how many bytes were mapped. Otherwise, the input and output values of *Length* are identical.

WriteToDevice

Indicates the direction of the transfer operation: TRUE for a transfer from the locked-down buffer to the device.

Include

wdm.h or *ntddk.h*

Return Value

MapTransfer returns the logical address of the region mapped, which the driver of a busmaster adapter can use. Drivers of devices that use a system DMA controller cannot use this value and should ignore it.

Comments

MapTransfer *is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

The *DmaAdapter* must have already been allocated as a result of the driver's preceding call to **AllocateAdapterChannel**.

The number of map registers that can be set up cannot exceed the maximum returned when the driver called **IoGetDmaAdapter**.

A driver can get the initial *CurrentVa* for the start of a packet-based DMA transfer by calling **MmGetMdlVirtualAddress**. However, the value returned is an index into the *Mdl*, rather than a valid virtual address. If the driver must split a large transfer request into more than one DMA operation, it must update *CurrentVa* and *Length* for each DMA operation.

The driver of a busmaster device with scatter/gather support can use the returned logical address and updated *Length* value to build a scatter/gather list, calling **MapTransfer** repeatedly until it has used all available map registers for the transfer operation. However, such a driver could more simply use the **GetScatterGatherList** routine.

Callers of **MapTransfer** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ADDRESS_AND_SIZE_TO_SPAN_PAGES, AllocateCommonBuffer, IoGetDmaAdapter, AllocateAdapterChannel, FlushAdapterBuffers, FreeAdapterChannel, FreeMapRegisters, KeFlushIoBuffers, MmGetMdlVirtualAddress

PutDmaAdapter

```
VOID
PutDmaAdapter(
  PDMA_ADAPTER DmaAdapter
  );
```

PutDmaAdapter frees a DMA_ADAPTER structure previously allocated by **IoGetDmaAdapter**.

Parameters

DmaAdapter

Points to the DMA_ADAPTER structure to be released.

Include

wdm.h or *ntddk.h*

Comments

PutDmaAdapter *is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

PutDmaAdapter frees a DMA adapter object previously allocated by **IoGetDmaAdapter**. Drivers should call **PutDmaAdapter** after completing DMA operations and freeing any map registers and common buffer allocated with this adapter object. After **PutDmaAdapter** returns, the driver can no longer use the DMA adapter object.

A driver must call **PutDmaAdapter** when it receives a PnP IRP_MN_STOP_DEVICE request.

Callers of **PutDmaAdapter** must be running at IRQL DISPATCH_LEVEL.

See Also

IoGetDmaAdapter, DMA_OPERATIONS

PutScatterGatherList

```
VOID
PutScatterGatherList(
  IN PDMA_ADAPTER DmaAdapter,
  IN PSCATTER_GATHER_LIST ScatterGather,
  IN BOOLEAN WriteToDevice
  );
```

PutScatterGatherList frees the previously allocated map registers and scatter/gather list used in scatter/gather DMA.

Parameters

DmaAdapter

Points to the DMA_ADAPTER structure returned by **IoGetDmaAdapter** that represents the busmaster adapter or DMA controller.

ScatterGather

Points to a scatter/gather list previously returned by **GetScatterGather**.

WriteToDevice

Indicates the direction of the DMA transfer: specify TRUE for a transfer from the buffer to the device, and FALSE otherwise.

Include

wdm.h or *ntddk.h*

Return Value

This routine can return the following NTSTATUS value:

Value	Meaning
STATUS_SUCCESS	The map registers and scatter/gather list were successfully deallocated.

Comments

PutScatterGatherList *is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a* DMA_OPERATIONS *structure.* Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

Drivers should call **PutScatterGatherList** after completing scatter/gather I/O. This routine flushes the adapter buffers, frees the map registers, and frees the scatter/gather list previously allocated by **GetScatterGatherList**.

Callers of **PutScatterGatherList** must be running at IRQL DISPATCH_LEVEL.

See Also

IoGetDmaAdapter, **GetScatterGatherList**, DMA_OPERATIONS, SCATTER_GATHER_LIST

ReadDmaCounter

```
ULONG
  ReadDmaCounter(
  IN PDMA_ADAPTER  DmaAdapter
  );
```

ReadDmaCounter returns the number of bytes remaining to be transferred during the current slave DMA operation.

Parameters

DmaAdapter

Points to the adapter object previously returned by **IoGetDmaAdapter** representing the system DMA controller channel currently in use.

Include

wdm.h or *ntddk.h*

Return Value

ReadDmaCounter returns the number of bytes remaining to be transferred in the current DMA operation.

Comments

ReadDmaCounter is not a system routine that can be called directly by name. This routine is callable only by pointer from the address returned in a DMA_OPERATIONS structure. Drivers obtain the address of this routine by calling **IoGetDmaAdapter**.

ReadDmaCounter can be called only by drivers of slave DMA devices. Usually, the caller is the driver of a slave device that uses a system DMA controller's autoinitialize mode.

Callers of **ReadDmaCounter** must be running at IRQL <= DISPATCH_LEVEL.

See Also

AllocateCommonBuffer, **IoGetDmaAdapter**, **FlushAdapterBuffers**, **MapTransfer**

READ_PORT_BUFFER_UCHAR

```
VOID
  READ_PORT_BUFFER_UCHAR(
  IN PUCHAR  Port,
  IN PUCHAR  Buffer,

IN ULONG  Count
  );
```

READ_PORT_BUFFER_UCHAR reads a number of bytes from the specified port address into a buffer.

Parameters

Port

Points to the port, which must be a mapped memory range in I/O space.

Buffer

Points to a buffer into which an array of UCHAR values is read.

Count

Specifies the number of bytes to be read into the buffer.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of READ_PORT_BUFFER_UCHAR can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

READ_PORT_BUFFER_ULONG

```
VOID
  READ_PORT_BUFFER_ULONG(
  IN PULONG  Port,
  IN PULONG  Buffer,
  IN ULONG  Count
  );
```

READ_PORT_BUFFER_ULONG reads a number of ULONG values from the specified port address into a buffer.

Parameters

Port

Points to the port, which must be a mapped memory range in I/O space.

Buffer

Points to a buffer into which an array of ULONG values is read.

Count

Specifies the number of ULONG values to be read into the buffer.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of READ_PORT_BUFFER_ULONG can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

READ_PORT_BUFFER_USHORT

```
VOID
  READ_PORT_BUFFER_USHORT(
  IN PUSHORT  Port,
  IN PUSHORT  Buffer,
  IN ULONG  Count
  );
```

READ_PORT_BUFFER_USHORT reads a number of USHORT values from the specified port address into a buffer.

Parameters

Port

Points to the port, which must be a mapped memory range in I/O space.

Buffer

Points to a buffer into which an array of USHORT values is read.

Count

Specifies the number of USHORT values to be read into the buffer.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of READ_PORT_BUFFER_USHORT can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

READ_PORT_UCHAR

```
UCHAR
  READ_PORT_UCHAR(
  IN PUCHAR  Port
  );
```

READ_PORT_UCHAR reads a byte from the specified port address.

Parameters

Port

Points to the port address, which must be a mapped memory range in I/O space.

Include

wdm.h or *ntddk.h*

Return Value

READ_PORT_UCHAR returns the byte read from the specified port address.

Comments

Callers of READ_PORT_UCHAR can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

READ_PORT_ULONG

```
ULONG
  READ_PORT_ULONG(
  IN PULONG  Port
  );
```

READ_PORT_ULONG reads a ULONG value from the specified port address.

Parameters

Port

Points to the port address, which must be a mapped range in I/O space.

Include

wdm.h or *ntddk.h*

Return Value

READ_PORT_ULONG returns the ULONG value read from the specified port address.

Comments

Callers of READ_PORT_ULONG can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

READ_PORT_USHORT

```
USHORT
  READ_PORT_USHORT(
  IN PUSHORT  Port
  );
```

READ_PORT_USHORT reads a USHORT value from the specified port address.

Parameters

Port

Points to the port address, which must be a mapped range in I/O space.

Include

wdm.h or *ntddk.h*

Return Value

READ_POR166_USHORT returns the USHORT value read from the specified port address.

Comments

Callers of READ_PORT_USHORT can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

READ_REGISTER_BUFFER_UCHAR

```
VOID
  READ_REGISTER_BUFFER_UCHAR(
  IN PUCHAR  Register,
  IN PUCHAR  Buffer,
  IN ULONG   Count
  );
```

READ_REGISTER_BUFFER_UCHAR reads a number of bytes from the specified register address into a buffer.

Parameters

Register

Points to the register, which must be a mapped range in memory space.

Buffer

Points to a buffer into which an array of UCHAR values is read.

Count

Specifies the number of bytes to be read into the buffer.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of READ_REGISTER_BUFFER_UCHAR can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

READ_REGISTER_BUFFER_ULONG

```
VOID
  READ_REGISTER_BUFFER_ULONG(
  IN PULONG  Register,
  IN PULONG  Buffer,
  IN ULONG   Count
  );
```

READ_REGISTER_BUFFER_ULONG reads a number of ULONG values from the specified register address into a buffer.

Parameters

Register

Points to the register, which must be a mapped range in memory space.

Buffer

Points to a buffer into which an array of ULONG values is read.

Count

Specifies the number of ULONG values to be read into the buffer.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of ULONG values.

Callers of READ_REGISTER_BUFFER_ULONG can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

READ_REGISTER_BUFFER_USHORT

```
VOID
  READ_REGISTER_BUFFER_USHORT(
  IN PUSHORT  Register,
  IN PUSHORT  Buffer,
  IN ULONG  Count
  );
```

READ_REGISTER_BUFFER_USHORT reads a number of USHORT values from the specified register address into a buffer.

Parameters

Register

Points to the register, which must be a mapped range in memory space.

Buffer

Points to a buffer into which an array of USHORT values is read.

Count

Specifies the number of USHORT values to be read into the buffer.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of USHORT values.

Callers of READ_REGISTER_BUFFER_USHORT can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

READ_REGISTER_UCHAR

```
UCHAR
  READ_REGISTER_UCHAR(
  IN PUCHAR  Register
  );
```

READ_REGISTER_UCHAR reads a byte from the specified register address.

Parameters

Register

Points to the register address, which must be a mapped range in memory space.

Include

wdm.h or *ntddk.h*

Return Value

READ_REGISTER_UCHAR returns the byte read from the specified register address.

Comments

Callers of READ_REGISTER_UCHAR can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

READ_REGISTER_ULONG

```
ULONG
  READ_REGISTER_ULONG(
  IN PULONG  Register
  );
```

READ_REGISTER_ULONG reads a ULONG value from the specified register address.

Parameters

Register

Points to the register address, which must be a mapped range in memory space.

Include

wdm.h or *ntddk.h*

Return Value

READ_REGISTER_ULONG returns the ULONG value read from the specified register address.

Comments

Callers of READ_REGISTER_ULONG can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

READ_REGISTER_USHORT

```
USHORT
  READ_REGISTER_USHORT(
  IN PUSHORT  Register
  );
```

READ_REGISTER_USHORT reads a USHORT value from the specified register address.

Parameters

Register

Points to the register address, which must be a mapped range in memory space.

Include

wdm.h or *ntddk.h*

Return Value

READ_REGISTER_USHORT returns the USHORT value read from the specified register address.

Comments

Callers of READ_REGISTER_USHORT can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

WRITE_PORT_BUFFER_UCHAR

```
VOID
  WRITE_PORT_BUFFER_UCHAR(
  IN PUCHAR  Port,
  IN PUCHAR  Buffer,
  IN ULONG  Count
  );
```

WRITE_PORT_BUFFER_UCHAR writes a number of bytes from a buffer to the specified port.

Parameters

Port

Points to the port, which must be a mapped memory range in I/O space.

Buffer

Points to a buffer from which an array of UCHAR values is to be written.

Count

Specifies the number of bytes to be written to the port.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of WRITE_PORT_BUFFER_UCHAR can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

WRITE_PORT_BUFFER_ULONG

```
VOID
  WRITE_PORT_BUFFER_ULONG(
  IN PULONG  Port,
  IN PULONG  Buffer,
  IN ULONG  Count
  );
```

WRITE_PORT_BUFFER_ULONG writes a number of ULONG values from a buffer to the specified port address.

Parameters

Port

Points to the port, which must be a mapped memory range in I/O space.

Buffer

Points to a buffer from which an array of ULONG values is to be written.

Count

Specifies the number of ULONG values to be written to the port.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of ULONGs.

Callers of WRITE_PORT_BUFFER_ULONG can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

WRITE_PORT_BUFFER_USHORT

```
VOID
  WRITE_PORT_BUFFER_USHORT(
  IN PUSHORT  Port,
  IN PUSHORT  Buffer,
  IN ULONG  Count
  );
```

WRITE_PORT_BUFFER_USHORT writes a number of USHORT values from a buffer to the specified port address.

Parameters

Port

Points to the port, which must be a mapped memory range in I/O space.

Buffer

Points to a buffer from which an array of USHORT values is to be written.

Count

Specifies the number of USHORT values to be written to the port.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of USHORTs.

Callers of WRITE_PORT_BUFFER_USHORT can be running at any IRQL, assuming the *Buffer* is resident and the *Port* is resident, mapped device memory.

WRITE_PORT_UCHAR

```
VOID
  WRITE_PORT_UCHAR(
  IN PUCHAR  Port,
  IN UCHAR  Value
  );
```

WRITE_PORT_UCHAR writes a byte to the specified port address.

Parameters

Port

Points to the port, which must be a mapped memory range in I/O space.

Value

Specifies a byte to be written to the port.

Include

wdm.h or *ntddk.h*

Comments

Callers of WRITE_PORT_UCHAR can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

WRITE_PORT_ULONG

```
VOID
  WRITE_PORT_ULONG(
  IN PULONG  Port,
  IN ULONG  Value
  );
```

WRITE_PORT_ULONG writes a ULONG value to the specified port address.

Parameters

Port

Points to the port, which must be a mapped memory range in I/O space.

Value

Specifies a ULONG value to be written to the port.

Include

wdm.h or *ntddk.h*

Comments

Callers of WRITE_PORT_ULONG can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

WRITE_PORT_USHORT

```
VOID
  WRITE_PORT_USHORT(
  IN PUSHORT  Port,
  IN USHORT  Value
  );
```

WRITE_PORT_USHORT writes a USHORT value to the specified port address.

Parameters

Port

Points to the port, which must be a mapped memory range in I/O space.

Value

Specifies a USHORT value to be written to the port.

Include

wdm.h or *ntddk.h*

Comments

Callers of WRITE_PORT_USHORT can be running at any IRQL, assuming the *Port* is resident, mapped device memory.

WRITE_REGISTER_BUFFER_UCHAR

```
VOID
  WRITE_REGISTER_BUFFER_UCHAR(
  IN PUCHAR  Register,
  IN PUCHAR  Buffer,
  IN ULONG  Count
  );
```

WRITE_REGISTER_BUFFER_UCHAR writes a number of bytes from a buffer to the specified register.

Parameters

Register

Points to the register, which must be a mapped range in memory space.

Buffer

Points to a buffer from which an array of UCHAR values is to be written.

Count

Specifies the number of bytes to be written to the register.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of bytes.

Callers of WRITE_REGISTER_BUFFER_UCHAR can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

WRITE_REGISTER_BUFFER_ULONG

```
VOID
  WRITE_REGISTER_BUFFER_ULONG(
  IN PULONG  Register,
  IN PULONG  Buffer,
  IN ULONG  Count
  );
```

WRITE_REGISTER_BUFFER_ULONG writes a number of ULONG values from a buffer to the specified register.

Parameters

Register

Points to the register, which must be a mapped range in memory space.

Buffer

Points to a buffer from which an array of ULONG values is to be written.

Count

Specifies the number of ULONG values to be written to the register.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of ULONGs.

Callers of WRITE_REGISTER_BUFFER_ULONG can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

WRITE_REGISTER_BUFFER_USHORT

```
VOID
  WRITE_REGISTER_BUFFER_USHORT(
  IN PUSHORT  Register,
  IN PUSHORT  Buffer,
  IN ULONG  Count
  );
```

WRITE_REGISTER_BUFFER_USHORT writes a number of USHORT values from a buffer to the specified register.

Parameters

Register

Points to the register, which must be a mapped range in memory space.

Buffer

Points to a buffer from which an array of USHORT values is to be written.

Count

Specifies the number of USHORT values to be written to the register.

Include

wdm.h or *ntddk.h*

Comments

The size of the buffer must be large enough to contain at least the specified number of USHORTs.

Callers of WRITE_REGISTER_BUFFER_USHORT can be running at any IRQL, assuming the *Buffer* is resident and the *Register* is resident, mapped device memory.

WRITE_REGISTER_UCHAR

```
VOID
  WRITE_REGISTER_UCHAR(
  IN PUCHAR  Register,
  IN UCHAR  Value
  );
```

WRITE_REGISTER_UCHAR writes a byte to the specified address.

Parameters

Register

Points to the register, which must be a mapped range in memory space.

Value

Specifies a byte to be written to the register.

Include

wdm.h or *ntddk.h*

Comments

Callers of WRITE_REGISTER_UCHAR can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

WRITE_REGISTER_ULONG

```
VOID
  WRITE_REGISTER_ULONG(
  IN PULONG  Register,
  IN ULONG  Value
  );
```

WRITE_REGISTER_ULONG writes a ULONG value to the specified address.

Parameters

Register

Points to the register which must be a mapped range in memory space.

Value

Specifies a ULONG value to be written to the register.

Include

wdm.h or *ntddk.h*

Comments

Callers of WRITE_REGISTER_ULONG can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

WRITE_REGISTER_USHORT

```
VOID
  WRITE_REGISTER_USHORT(
  IN PUSHORT  Register,
  IN USHORT  Value
  );
```

WRITE_REGISTER_USHORT writes a USHORT value to the specified address.

Parameters

Register

Points to the register, which must be a mapped range in memory space.

Value

Specifies a USHORT value to be written to the register.

Include

wdm.h or *ntddk.h*

Comments

Callers of WRITE_REGISTER_USHORT can be running at any IRQL, assuming the *Register* is resident, mapped device memory.

CHAPTER 4

I/O Manager Routines

All kernel-mode drivers except video and SCSI miniport drivers and NDIS drivers call **Io***Xxx* routines.

References for the **Io***Xxx* routines are in alphabetical order.

For an overview of the functionality of these routines, see Chapter 1, *Summary of Kernel-Mode Support Routines*.

IoAcquireCancelSpinLock

```
VOID
  IoAcquireCancelSpinLock(
  OUT PKIRQL  Irql
  );
```

IoAcquireCancelSpinLock synchronizes cancelable-state transitions for IRPs in a multiprocessor-safe way.

Parameters

Irql

Points to a variable in which to save the current IRQL for a subsequent call to **IoReleaseCancelSpinLock**. Usually, the *Irql* is saved on the stack as a local variable.

Include

wdm.h or *ntddk.h*

Comments

A driver that uses the I/O-manager-supplied device queues in the device object must be holding the cancel spin lock whenever it changes the cancelable state of an IRP with **IoSetCancelRoutine**.

A driver that manages its own queue(s) of IRPs does not need to hold the cancel spin lock when calling **IoSetCancelRoutine**.

The holder of the cancel spin lock should release it promptly by calling **IoReleaseCancelSpinLock**.

A driver-supplied Cancel routine is called with the cancel spin lock held. It must release the cancel spin lock when it has completed the IRP to be canceled.

Callers of **IoAcquireCancelSpinLock** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoReleaseCancelSpinlock, IoSetCancelRoutine

IoAcquireRemoveLock

This routine is documented in Volume 1 of the *Windows 2000 Drivers Development Reference*. Please see ***IoAcquireRemoveLock*** in that book for a full reference.

IoAcquireRemoveLockEx

This routine is documented in Volume 1 of the *Windows 2000 Drivers Development Reference*. Please see ***IoAcquireRemoveLock*** in that book for a full reference.

IoAdjustPagingPathCount

This routine is documented in Volume 1 of the *Windows 2000 Drivers Development Reference*. Please see ***IoAcquireRemoveLock*** in that book for a full reference.

IoAllocateAdapterChannel

```
NTSTATUS
  IoAllocateAdapterChannel(
  IN PADAPTER_OBJECT  AdapterObject,
  IN PDEVICE_OBJECT  DeviceObject,
  IN ULONG  NumberOfMapRegisters,
  IN PDRIVER_CONTROL  ExecutionRoutine,
  IN PVOID  Context
  );
```

IoAllocateAdapterChannel is obsolete and is exported only to support existing drivers. Use **AllocateAdapterChannel** instead.

Return Value

IoAllocateWorkItem returns a pointer to a private IO_WORKITEM structure. Drivers should not make any assumptions about the format of this structure nor attempt to access information contained in this structure. **IoAllocateWorkItem** can return NULL in the case of insufficient resources.

Comments

Drivers queue work items allocated by **IoAllocateWorkItem** with **IoQueueWorkItem**.

It is the caller's responsibility to free the resources associated with the work item returned by **IoAllocateWorkItem** by calling **IoFreeWorkItem** in the callback routine passed to **IoQueueWorkItem**.

Callers of **IoAllocateWorkItem** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoQueueWorkItem, **IoFreeWorkItem**

IoAssignArcName

```
VOID
  IoAssignArcName(
  IN PUNICODE_STRING  ArcName,
  IN PUNICODE_STRING  DeviceName
  );
```

IoAssignArcName creates a symbolic link between the ARC name of a physical device and the name of the corresponding device object when it has been created.

Parameters

ArcName

Points to a buffer containing the ARC name of the device. The ARC name must be a Unicode string.

DeviceName

Points to a buffer containing the name of the device object, representing the same device. The device object name must be a Unicode string.

Include

ntddk.h

Comments

Drivers of hard disk devices need not call this routine. Drivers of other mass-storage devices, including floppy, CD_ROM, and tape devices, should call **IoAssignArcName** during their initialization.

Callers of **IoAssignArcName** must be running at IRQL PASSIVE_LEVEL.

See Also

IoCreateDevice

IoAssignResources

```
NTSTATUS
  IoAssignResources(
  IN PUNICODE_STRING  RegistryPath,
  IN PUNICODE_STRING  DriverClassName  OPTIONAL,
  IN PDRIVER_OBJECT  DriverObject,
  IN PDEVICE_OBJECT  DeviceObject  OPTIONAL,
  IN PIO_RESOURCE_REQUIREMENTS_LIST  RequestedResources,
  IN OUT PCM_RESOURCE_LIST  *AllocatedResources
  );
```

IoAssignResources is *obsolete* and is exported only to support existing drivers.

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.

IoAssignResources takes an input list of requested hardware resources for a driver or device, claims an available set of hardware resources, such as an interrupt vector, device memory range and/or I/O port range, and possibly a particular DMA controller channel, in the \Registry\Hardware\Machine\ResourceMap tree, and returns a list of allocated hardware resources for the driver or device. As an alternative, drivers of PCI-type devices can call HalAssignSlotResources.

Parameters

RegistryPath

Points to the **\Registry\Machine\System\CurrentControlSet\Services***DriverName* key or one of its subkeys, depending on whether the input *DeviceObject* pointer is NULL. If a driver uses resources in common for all its devices, *RegistryPath* is the pointer input to its **DriverEntry** routine and the *DeviceObject* pointer must be NULL. A driver that needs device-specific hardware resources, rather than driver-specific resources in common for all

its devices, must pass a *RegistryPath* pointer to an updated, device-specific string naming a subkey of *DriverName*, at each call to **IoAssignResources** with a nonNULL pointer to a unique *DeviceObject*.

DriverClassName

Points to a buffered Unicode string that describes the class of driver under which the driver's configuration information should be stored. A default type **Other** is used if none is given, and a new key is created in the registry if a unique name is supplied.

DriverObject

Points to the driver object that was input to the DriverEntry routine.

DeviceObject

This pointer is optional. If it is NULL, the caller-supplied *RequestedResources* list specifies resources that the driver itself needs, possibly to control several devices that it supports. Otherwise, *DeviceObject* points to the driver-created device object representing a physical device for which the driver is attempting to claim device-specific hardware resources.

RequestedResources

Points to a caller-supplied IO_RESOURCE_REQUIREMENTS_LIST structure. This structure contains a list of raw hardware resources needed by one or more devices, which the driver has found by calling **HalGetBusData** or **HalGetBusDataByOffset**, by interrogating its devices, or by some other means. The driver can allocate the structure from paged memory.

AllocatedResources

Points to the address of a location to receive a pointer to a CM_RESOURCE_LIST structure, which describes the raw hardware resources allocated for the caller. The caller is responsible for freeing the buffer.

Include

ntddk.h

Return Value

IoAssignResources returns STATUS_SUCCESS if it claimed a set of the specified hardware resources for the caller and returned information in the *AllocatedResources* buffer. Otherwise, it returns an error status, resets the pointer at *AllocatedResources* to NULL, and logs an error if it finds a resource conflict.

Comments

For most device drivers, calling **IoAssignResources** after locating the device and getting whatever configuration information **HalGetBusData** or **HalGetBusDataByOffset** can

supply is preferable to making paired calls to **IoQueryDeviceDescription** and **IoReportResourceUsage**.

Note that **IoAssignResources** does not handle IO_RESOURCE_DESCRIPTOR entries with the **Type** member set to **CmResourceTypeDeviceSpecific**. Drivers that have hardware resources of this type can call **IoReportResourceUsage** to store this configuration information in the **\\Registry\..\ResourceMap** tree. Otherwise, a successful call to **IoAssignResources** writes the caller's claims on every other type of hardware resource into the registry **\ResourceMap** tree.

A driver can supply any number of IO_RESOURCE_LIST elements, each containing IO_RESOURCE_DESCRIPTOR structures specifying both preferred and alternative hardware resources the driver can use, if the device or I/O bus does not constrain the driver to using a fixed range of I/O ports or device memory, a fixed bus-specific interrupt vector, and/or a particular DMA channel or port number. In particular, drivers of devices that can be configured to use alternate sets of hardware resources are expected to take advantage of this capability, although drivers of PCI-type devices can call **HalAssignSlotResources** instead. If **IoAssignResources** cannot claim a preferred set of resources, it tries an alternative set and returns the set of resources claimed as soon as it can satisfy the request with a given alternate resource list.

IoAssignResources automatically searches the registry for resource conflicts between resources requested and resources claimed by previously installed drivers. It first matches the preferred entries in the *RequestedResources* descriptor array against all other resource lists stored in the registry to determine whether a conflict exists. If it finds a conflict, it then matches any supplied alternative descriptors for the already claimed resource again, attempting to allocate a set of resources the caller can use.

The caller is responsible for releasing the *AllocatedResources* buffer, which is pageable, with **ExFreePool** after it has consumed the returned information and before the DriverEntry routine returns control.

If a driver claims resources on a device-specific basis for more than one device, the driver must call this routine at least once for each such device, and must update the *RegistryPath* string to supply a unique subkey name for each call with a unique *DeviceObject* pointer.

This routine can be called more than once for a given device or driver. If a new list of *RequestedResources* is supplied, it will overwrite or, possibly, be appended to the previous resource list in the registry. However, making a single call for each set of device-specific resources makes a driver load much faster than if it calls **IoAssignResources** many times to amend or incrementally construct the input *RequestedResources* for each of its devices. Note that subsequent calls to **IoAssignResources** can reassign the caller's previously claimed resources if that caller does not adjust the input *RequestedReources* to "fix" its claim on the resources to be kept.

A driver must call **IoAssignResources** with a value of NULL for the *RequestedResources* parameter to erase its claim on resources in the registry if the driver is unloaded.

Callers of **IoAssignResources** must be running at IRQL PASSIVE_LEVEL.

See Also

IRP_MN_START_DEVICE, **IoReportDetectedDevice**, **IoReportResourceForDetection**

IoAttachDevice

```
NTSTATUS
  IoAttachDevice(
  IN PDEVICE_OBJECT  SourceDevice,
  IN PUNICODE_STRING  TargetDevice,
  OUT PDEVICE_OBJECT  *AttachedDevice
  );
```

IoAttachDevice attaches the caller's device object to a named target device object, so that I/O requests bound for the target device are routed first to the caller.

Parameters

SourceDevice

Points to the caller-created device object.

TargetDevice

Points to a buffer containing the name of the device object to which the specified *Source-Device* is to be attached.

AttachedDevice

Points to caller-allocated storage for a pointer. On return, contains a pointer to the target device object if the attachment succeeds.

Include

wdm.h or *ntddk.h*

Return Value

IoAttachDevice can return one of the following NTSTATUS values:

STATUS_SUCCESS
STATUS_INVALID_PARAMETER
STATUS_OBJECT_TYPE_MISMATCH
STATUS_OBJECT_NAME_INVALID
STATUS_INSUFFICIENT_RESOURCES

Comments

IoAttachDevice establishes layering between drivers so that the same IRPs can be sent to each driver in the chain.

This routine is used by intermediate drivers during initialization. It allows such a driver to attach its own device object to another device in such a way that any requests being made to the original device are given first to the intermediate driver.

The caller can be layered only at the top of an existing chain of layered drivers. **IoAttachDevice** searches for the highest device object layered over *TargetDevice* and attaches to that object (that can be the *TargetDevice*). Therefore, this routine must not be called if a driver that must be higher-level has already layered itself over the target device.

Note that for file system drivers and drivers in the storage stack, **IoAttachDevice** opens the target device with FILE_READ_ATTRIBUTES and then calls **IoGetRelatedDeviceObject**. This does not cause a file system to be mounted. Thus, a successful call to **IoAttachDevice** returns the device object of the storage driver, not that of the file system driver.

This routine sets the **AlignmentRequirement** in *SourceDevice* to the value in the next-lower device object and sets the **StackSize** to the value in the next-lower-object plus one.

Callers of **IoAttachDevice** must be running at IRQL PASSIVE_LEVEL.

See Also

IoAttachDeviceToDeviceStack, **IoGetRelatedDeviceObject**, **IoCreateDevice**, **IoDetachDevice**

IoAttachDeviceByPointer

```
NTSTATUS
  IoAttachDeviceByPointer(
  IN PDEVICE_OBJECT  SourceDevice,
  IN PDEVICE_OBJECT  TargetDevice
  );
```

This routine is obsolete; use **IoAttachDeviceToDeviceStack**.

IoAttachDeviceToDeviceStack

```
PDEVICE_OBJECT
  IoAttachDeviceToDeviceStack(
  IN PDEVICE_OBJECT  SourceDevice,
  IN PDEVICE_OBJECT  TargetDevice
  );
```

IoAttachDeviceToDeviceStack attaches the caller's device object to the highest device object in the chain and returns a pointer to the previously highest device object. I/O requests bound for the target device are routed first to the caller.

Parameters

SourceDevice

Points to the caller-created device object.

TargetDevice

Points to another driver's device object, such as a pointer returned by a preceding call to **IoGetDeviceObjectPointer**.

Include

wdm.h or *ntddk.h*

Return Value

IoAttachDeviceToDeviceStack returns a pointer to the device object to which the *SourceDevice* was attached. The returned device object pointer can differ from *TargetDevice* if *TargetDevice* had additional drivers layered on top of it.

IoAttachDeviceToDeviceStack returns NULL if it could not attach the device object because, for example, the target device was being unloaded.

Comments

IoAttachDeviceToDeviceStack establishes layering between drivers so that the same IRPs are sent to each driver in the chain.

An intermediate driver can use this routine during initialization to attach its own device object to another driver's device object. Subsequent I/O requests sent to *TargetDevice* are sent first to the intermediate driver.

This routine sets the **AlignmentRequirement** in *SourceDevice* to the value in the next-lower device object and sets the **StackSize** to the value in the next-lower-object plus one.

A driver writer must take care to call this routine *before* any drivers that must layer on top of their driver. **IoAttachDeviceToDeviceStack** attaches *SourceDevice* to the highest device object currently layered in the chain and has no way to determine whether drivers are being layered in the correct order.

A driver that acquired a pointer to the target device by calling **IoGetDeviceObjectPointer** should call **ObDereferenceObject** with the file object pointer that was returned by **IoGetDeviceObjectPointer** to release its reference to the file object before it detaches its own device object, for example, when such a higher-level driver is unloaded.

Callers of **IoAttachDeviceToDeviceStack** must be running at IRQL PASSIVE_LEVEL.

See Also

IoAttachDevice, IoDetachDevice, ObDereferenceObject, IoGetDeviceObjectPointer

IoBuildAsynchronousFsdRequest

```
PIRP
  IoBuildAsynchronousFsdRequest(
  IN ULONG  MajorFunction,
  IN PDEVICE_OBJECT  DeviceObject,
  IN OUT PVOID  Buffer  OPTIONAL,
  IN ULONG  Length  OPTIONAL,
  IN PLARGE_INTEGER  StartingOffset  OPTIONAL,
  IN PIO_STATUS_BLOCK  IoStatusBlock  OPTIONAL
  );
```

IoBuildAsynchronousFsdRequest allocates and sets up an IRP to be sent to lower-level drivers.

Parameters

MajorFunction

Specifies the major function code to be set in the IRP, one of IRP_MJ_PNP, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_FLUSH_BUFFERS, or IRP_MJ_SHUTDOWN.

DeviceObject

Points to the next-lower driver's device object, representing the target device for the read, write, flush, or shutdown operation.

Buffer

Points to a buffer into which data is read or from which data is written. The value of this argument is NULL for flush and shutdown requests.

Length

Specifies the length in bytes of *Buffer*. The value of this argument is zero for flush and shutdown requests.

StartingOffset

Points to the starting offset on the input/output media. The value of this argument is zero for flush and shutdown requests.

IoStatusBlock

Points to the address of an I/O status block in which the to-be-called driver(s) return final status about the requested operation.

Include

wdm.h or *ntddk.h*

Return Value

IoBuildAsynchronousFsdRequest returns a pointer to an IRP or a NULL pointer if the IRP cannot be allocated.

Comments

Intermediate or highest-level drivers can call **IoBuildAsynchronousFsdRequest** to set up IRPs for requests sent to lower-level drivers. Such a driver must set its IoCompletion routine in the IRP so the IRP can be deallocated with **IoFreeIrp**.

The IRP that gets built contains only enough information to get the operation started and to complete the IRP. No other context information is tracked because an asynchronous request is context-independent.

Callers of **IoBuildAsynchronousFsdRequest** must be running at IRQL <= DISPATCH_LEVEL.

An intermediate or highest-level driver also can call **IoBuildDeviceIoControlRequest**, **IoAllocateIrp**, or **IoBuildSychronousFsdRequest** to set up requests it sends to lower-level drivers. Only a highest-level driver can call **IoMakeAssociatedIrp**.

See Also

IO_STACK_LOCATION, **IoAllocateIrp**, **IoBuildDeviceIoControlRequest**, **IoBuildSynchronousFsdRequest**, **IoCallDriver**, **IoFreeIrp**, **IoMakeAssociatedIrp**, **IoSetCompletionRoutine**, IRP

IoBuildDeviceIoControlRequest

```
PIRP
  IoBuildDeviceIoControlRequest(
  IN ULONG  IoControlCode,
  IN PDEVICE_OBJECT  DeviceObject,
  IN PVOID  InputBuffer  OPTIONAL,
  IN ULONG  InputBufferLength,
  OUT PVOID  OutputBuffer  OPTIONAL,
  IN ULONG  OutputBufferLength,
  IN BOOLEAN  InternalDeviceIoControl,
```

```
IN PKEVENT  Event,
OUT PIO_STATUS_BLOCK  IoStatusBlock
);
```

IoBuildDeviceIoControlRequest allocates and sets up an IRP for a device control request, optionally with an I/O buffer if the I/O control code requires the caller to supply an input or output buffer.

Parameters

IoControlCode

Specifies the IOCTL_*XXX* to be set up. For more information about device-type-specific I/O codes, see Part 2 of this volume.

DeviceObject

Points to the next-lower driver's device object, representing the target device.

InputBuffer

Points to an input buffer to be passed to the lower driver or NULL if the request does not pass input data to lower driver(s).

InputBufferLength

Specifies the length in bytes of the input buffer. If *InputBuffer* is NULL, this value must be zero.

OutputBuffer

Points to an output buffer in which the lower driver is to return data or NULL if the request does not require lower driver(s) to return data.

OutputBufferLength

Specifies the length in bytes of the output buffer. If *OutputBuffer* is NULL, this value must be zero.

InternalDeviceIoControl

If *InternalDeviceControl* is TRUE the target driver's Dispatch routine for IRP_MJ_INTERNAL_DEVICE_CONTROL or IRP_MJ_SCSI is called; otherwise, the Dispatch routine for IRP_MJ_DEVICE_CONTROL is called.

Event

Points to an initialized event object for which the caller provides the storage. The event is set to the Signaled state when lower driver(s) have completed the requested operation. The caller can wait on the event object for the completion of the IRP allocated by this routine.

IoStatusBlock

Specifies an I/O status block to be set when the request is completed by lower drivers.

Include

wdm.h or *ntddk.h*

Return Value

IoBuildDeviceIoControlRequest returns a pointer to an IRP with the next-lower driver's I/O stack location partially set up from the supplied parameters. The returned pointer is NULL if an IRP cannot be allocated.

Comments

An intermediate or highest-level driver can call **IoBuildDeviceIoControlRequest** to set up IRPs for requests sent to lower-level drivers. The next-lower driver's I/O stack location is set up with the given *IoControlCode* at **Parameters.DeviceIoControl.IoControlCode**. Because the caller can wait on the completion of this driver-allocated IRP by calling **KeWaitForSingleObject** on the given *Event*, the caller need not set an IoCompletion routine in the IRP before calling **IoCallDriver**. When the next-lower driver completes this IRP, the I/O Manager releases it.

IRPs created using **IoBuildDeviceIoControlRequest** must be completed by calling **IoCompleteRequest** and not by merely deallocating the IRP with **IoFreeIrp**. **IoBuildDeviceIoControlRequest** queues the IRPs it creates in the IRP queue of the current thread. Freeing these IRPs without completing them might result in a system crash when the thread terminates as the thread attempts to deallocate the IRP's memory.

Callers of **IoBuildDeviceIoControlRequest** must be running at IRQL PASSIVE_LEVEL.

See Also

IO_STACK_LOCATION, **IoAllocateIrp**, **IoBuildAsynchronousFsdRequest**, **IoBuildSynchronousFsdRequest**, **IoCallDriver**, **IoCompleteRequest**, IRP, **KeInitializeEvent**, **KeWaitForSingleObject**

IoBuildPartialMdl

```
VOID
  IoBuildPartialMdl(
  IN PMDL  SourceMdl,
  IN OUT PMDL  TargetMdl,
  IN PVOID  VirtualAddress,
  IN ULONG  Length
  );
```

IoBuildPartialMdl maps a portion of a buffer described by another MDL into an MDL.

Parameters

SourceMdl

Points to an MDL describing the original buffer, of which a subrange is to be mapped.

TargetMdl

Points to a caller-allocated MDL. The MDL must be large enough to map the subrange specified by *VirtualAddress* and *Length*.

VirtualAddress

Points to the base virtual address for the subrange to be mapped in the *TargetMdl*.

Length

Specifies the length in bytes to be mapped by the *TargetMdl*. This value, in combination with *VirtualAddress*, must specify a buffer that is a proper subrange of the buffer described by *SourceMdl*. If *Length* is zero, the subrange to be mapped starts at *VirtualAddress* and includes the remaining range described by the *SourceMdl*.

Include

wdm.h or *ntddk.h*

Comments

IoBuildPartialMdl maps a subrange of a buffer currently mapped by *SourceMdl*. The *VirtualAddress* and *Length* parameters describe the subrange to be mapped from the *SourceMdl* into the *TargetMdl*.

Drivers that must split large transfer requests can use this routine. The caller must release the partial MDL it allocated when it has transferred all the requested data or completed the IRP with an error status.

Callers of **IoBuildPartialMdl** can be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateMdl, **IoCallDriver**, **IoFreeMdl**, **IoSetCompletionRoutine**

IoBuildSynchronousFsdRequest

```
PIRP
  IoBuildSynchronousFsdRequest(
  IN ULONG  MajorFunction,
  IN PDEVICE_OBJECT  DeviceObject,
  IN OUT PVOID  Buffer  OPTIONAL,
  IN ULONG  Length  OPTIONAL,
  IN PLARGE_INTEGER  StartingOffset  OPTIONAL,
  IN PKEVENT  Event,
  OUT PIO_STATUS_BLOCK  IoStatusBlock
  );
```

IoBuildSynchronousFsdRequest allocates and builds an IRP to be sent synchronously to lower driver(s).

Parameters

MajorFunction

Specifies the major function code, one of IRP_MJ_PNP, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_FLUSH_BUFFERS, or IRP_MJ_SHUTDOWN.

DeviceObject

Points to the next-lower driver's device object representing the target device for the read, write, flush, or shutdown operation.

Buffer

Points to a buffer containing data to be written when *MajorFunction* is IRP_MJ_WRITE, or is the location to receive data read when *MajorFunction* is IRP_MJ_READ. This parameter must be NULL for the *MajorFunction* IRP_MJ_FLUSH_BUFFERS or IRP_MJ_SHUTDOWN.

Length

Specifies the length, in bytes, of *Buffer*. For devices such as disks, this value must be an integral of 512. This parameter is required for read/write requests, but must be zero for flush and shutdown requests.

StartingOffset

Points to the offset on the disk to read/write from/to. This parameter is required for read/write requests, but must be zero for flush and shutdown requests.

Event

Points to an initialized event object for which the caller provides the storage. The event is set to the Signaled state when the requested operation completes. The caller can wait on the event object for the completion of the IRP allocated by this routine.

IoStatusBlock

Points to the I/O status block that is set when the IRP is completed by the lower driver(s).

Include

wdm.h or *ntddk.h*

Return Value

IoBuildSynchronousFsdRequest returns a pointer to the IRP or NULL if an IRP cannot be allocated.

Comments

Intermediate or highest-level drivers can call **IoBuildSynchronousFsdRequest** to set up IRPs for requests sent to lower-level drivers, only if the caller is running in a nonarbitrary thread context and at IRQL PASSIVE_LEVEL.

IoBuildSynchronousFsdRequest allocates and sets up an IRP that can be sent to a device driver to perform a synchronous read, write, flush, or shutdown operation. The IRP contains only enough information to get the operation started.

The caller can determine when the I/O has completed by calling **KeWaitForSingleObject** with the *Event*. Performing this wait operation causes the current thread to wait. Therefore, this operation can be requested during the initialization of an intermediate driver or from an FSD in the context of a thread requesting a synchronous I/O operation. A driver cannot wait for a nonzero interval on the *Event* at raised IRQL in an arbitrary thread context.

Because the caller can wait on a given *Event*, the caller need not set an IoCompletion routine in the caller-allocated IRP before calling **IoCallDriver**. When the caller completes the IRP, the I/O Manager releases it.

IRPs created using **IoBuildSynchronousFsdRequest** must be completed by calling **IoCompleteRequest** and not by merely deallocating the IRP with **IoFreeIrp**. **IoBuild-SynchronousFsdRequest** queues the IRPs it creates in the IRP queue of the current thread. Freeing these IRPs without completing them might result in a system crash when the thread terminates as the thread attempts to deallocate the IRP's memory.

See Also

IO_STACK_LOCATION, **IoAllocateIrp**, **IoBuildAsynchronousFsdRequest**, **Io-CompleteRequest**, IRP, **KeInitializeEvent**, **KeWaitForSingleObject**

IoCallDriver

```
NTSTATUS
  IoCallDriver(
  IN PDEVICE_OBJECT  DeviceObject,
  IN OUT PIRP  Irp
  );
```

IoCallDriver sends an IRP to the next-lower-level driver after the caller has set up the I/O stack location in the IRP for that driver.

Parameters

DeviceObject

Points to the next-lower driver's device object, representing the target device for the requested I/O operation.

Irp

Points to the IRP.

Include

wdm.h or *ntddk.h*

Return Value

IoCallDriver returns the NTSTATUS value that a lower driver set in the I/O status block for the given request or STATUS_PENDING if the request was queued for additional processing.

Comments

IoCallDriver assigns the *DeviceObject* input parameter to the device object field of the IRP stack location for the next lower driver.

An IRP passed in a call to **IoCallDriver** becomes inaccessible to the higher-level driver, unless the higher-level driver has set up its IoCompletion routine for the IRP with **IoSetCompletionRoutine**. If it does, the IRP input to the driver-supplied IoCompletion routine has its I/O status block set by the lower driver(s) and all lower-level driver(s)' I/O stack locations filled with zeros.

Drivers must not use **IoCallDriver** to pass power IRPs (IRP_MJ_POWER). Use **PoCallDriver** instead.

Callers of **IoCallDriver** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateIrp, IoBuildAsynchronousFsdRequest, IoBuildDeviceIoControlRequest, IoBuildSynchronousFsdRequest, IoSetCompletionRoutine, PoCallDriver

IoCancelIrp

```
BOOLEAN
  IoCancelIrp(
  IN PIRP  Irp
  );
```

IoCancelIrp sets the cancel bit in a given IRP and calls the cancel routine for the IRP if there is one.

Parameters

Irp

Points to the IRP to be canceled.

Include

wdm.h or *ntddk.h*

Return Value

IoCancelIrp returns TRUE if the IRP was canceled and FALSE if the IRP's cancel bit was set but the IRP was not cancelable.

Comments

If the IRP has a cancel routine, **IoCancelIrp** sets the cancel bit and calls the cancel routine.

If **Irp->CancelRoutine** is NULL, and therefore the IRP is not cancelable, **IoCancelIrp** sets the IRP's cancel bit and returns FALSE. The IRP should be canceled at a later time when it becomes cancelable.

If a driver that does not own the IRP calls **IoCancelIrp**, the results are unpredictable. The IRP might be completed with a successful status even though its cancel bit was set.

An intermediate driver should not arbitrarily call **IoCancelIrp** unless that driver created the IRP passed in the call. Otherwise, the intermediate driver might cancel an IRP that some higher-level driver is tracking for purposes of its own.

Callers of **IoCancelIrp** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoSetCancelRoutine

IoCheckShareAccess

```
NTSTATUS
  IoCheckShareAccess(
  IN ACCESS_MASK  DesiredAccess,
  IN ULONG  DesiredShareAccess,
  IN OUT PFILE_OBJECT  FileObject,
  IN OUT PSHARE_ACCESS  ShareAccess,
  IN BOOLEAN  Update
  );
```

IoCheckShareAccess is called by FSDs or other highest-level drivers to check whether shared access to a file object is permitted.

Parameters

DesiredAccess

Specifies the desired type(s) of access to the given *FileObject* for the current open request. Generally, the value of this parameter is equal to the *DesiredAccess* passed to the file system or highest-level driver by the I/O Manager when the open request was made. See ***IoCreateFile*** for details.

DesiredShareAccess

Specifies the desired type(s) of shared access to *FileObject* for the current open request. The value of this parameter is usually the same as the *DesiredAccess* passed to the file system or highest-level driver by the I/O Manager when the open request was made. This value can be zero, one, or more of the following:

FILE_SHARE_READ
FILE_SHARE_WRITE
FILE_SHARE_DELETE

FileObject

Points to the file object for which to check access for the current open request.

ShareAccess

Points to the common share-access data structure associated with *FileObject*. Drivers should treat this structure as opaque.

Update

Specifies whether to update the share-access status for *FileObject*. A Boolean value of TRUE means this routine will update the share access information for the file object if the open request is permitted.

Include

wdm.h or *ntddk.h*

Return Value

IoCheckShareAccess returns STATUS_SUCCESS if the requestor's access to the file object is compatible with the way in which it is currently open. If the request is denied because of a sharing violation, then STATUS_SHARING_VIOLATION is returned.

Comments

IoCheckShareAccess checks a file object open request to determine whether the types of desired and shared accesses specified are compatible with the way in which the file object is currently being accessed by other opens.

File systems maintain state about files through structures called file control blocks (FCBs). The SHARE_ACCESS is a structure describing how the file is currently accessed by all opens. This state is contained in the FCB as part of the open state for each file object. Each file object should have only one share access structure. Other highest-level drivers might call this routine to check the access requested when a file object representing such a driver's device object is opened.

IoCheckShareAccess is not an atomic operation. Therefore, drivers calling this routine must protect the shared file object passed to **IoCheckShareAccess** by means of some kind of lock, such as a mutex or a resource lock, in order to prevent corruption of the shared access counts.

Callers of **IoCheckShareAccess** must be running at IRQL PASSIVE_LEVEL.

See Also

IoCreateFile, **IoGetRelatedDeviceObject**, **IoRemoveShareAccess**, **IoSetShareAccess**, **IoUpdateShareAccess**

IoCompleteRequest

```
VOID
  IoCompleteRequest(
  IN PIRP  Irp,
  IN CCHAR  PriorityBoost
  );
```

IoCompleteRequest indicates the caller has completed all processing for a given I/O request and is returning the given IRP to the I/O Manager.

Parameters

Irp

Points to the IRP to be completed.

PriorityBoost

Specifies a system-defined constant by which to increment the runtime priority of the original thread that requested the operation. This value is IO_NO_INCREMENT if the original thread requested an operation the driver could complete quickly (so the requesting thread is not compensated for its assumed wait on I/O) or if the IRP is completed with an error. Otherwise, the set of *PriorityBoost* constants are device-type-specific. See *ntddk.h* or *wdm.h* for these constants.

Include

wdm.h or *ntddk.h*

Comments

When a driver has finished all processing for a given IRP, it calls **IoCompleteRequest**. The I/O Manager checks the IRP to determine whether any higher-level drivers have set up an IoCompletion routine 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 a given IRP, the I/O Manger returns status to the original requestor 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.

Callers of **IoCompleteRequest** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoSetCompletionRoutine

IoConnectInterrupt

```
NTSTATUS
  IoConnectInterrupt(
  OUT PKINTERRUPT  *InterruptObject,
  IN PKSERVICE_ROUTINE  ServiceRoutine,
  IN PVOID  ServiceContext,
  IN PKSPIN_LOCK  SpinLock  OPTIONAL,
  IN ULONG  Vector,
  IN KIRQL  Irql,
  IN KIRQL  SynchronizeIrql,
  IN KINTERRUPT_MODE  InterruptMode,
  IN BOOLEAN  ShareVector,
```

```
IN KAFFINITY  ProcessorEnableMask,
IN BOOLEAN  FloatingSave
);
```

IoConnectInterrupt registers a device driver's interrupt service routine (ISR) to be called when its device interrupts on any of a given set of processors.

Parameters

InterruptObject

Points to the address of driver-supplied storage for a pointer to a set of interrupt objects. This pointer must be passed in subsequent calls to **KeSynchronizeExecution**.

ServiceRoutine

Points to the entry point for the driver-supplied ISR declared as follows:

```
BOOLEAN
(*PKSERVICE_ROUTINE)(
    IN PKINTERRUPT Interrupt,
    IN PVOID ServiceContext
    );
```

ServiceContext

Points to the driver-determined context with which the specified ISR will be called. The *ServiceContext* area must be in resident memory: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the device driver. See *Basic ISR Functionality* in Chapter 8 of the Kernel-Mode Drivers Design Guide for details.

SpinLock

Points to an initialized spin lock, for which the driver supplies the storage, that will be used to synchronize access to driver-determined data shared by other driver routines. This parameter is required if the ISR handles more than one vector or if the driver has more than one ISR. Otherwise, the driver need not allocate storage for an interrupt spin lock and the input pointer is NULL.

Vector

Specifies the interrupt vector passed in the interrupt resource at **u.Interrupt.Vector**.

Irql

Specifies the DIRQL passed in the interrupt resource at **u.Interrupt.Level**.

SynchronizeIrql

Specifies the DIRQL at which the ISR will execute. If the ISR handles more than one interrupt vector or the driver has more than one ISR, this value must be the highest of the

Irql values passed at **u.Interrupt.Level** in each interrupt resource. Otherwise, the *Irql* and *SynchronizeIrql* values are identical.

InterruptMode

Specifies whether the device interrupt is **LevelSensitive** or **Latched**.

ShareVector

Specifies whether the interrupt vector is sharable.

ProcessorEnableMask

Specifies the set of processors on which device interrupts can occur in this platform. This value is passed in the interrupt resource at **u.Interrupt.Affinity**.

FloatingSave

Specifies whether to save the floating-point stack when the driver's device interrupts. For X86-based platforms, this value must be set to FALSE.

Include

wdm.h or *ntddk.h*

Return Value

IoConnectInterrupt can return one of the following NTSTATUS values:

STATUS_SUCCESS
STATUS_INVALID_PARAMETER
STATUS_INSUFFICIENT_RESOURCES

Comments

A PnP driver should call **IoConnectInterrupt** as part of device start-up, before it completes the PnP IRP_MN_START_DEVICE request.

A driver receives raw and translated hardware resources with the IRP_MN_START_DEVICE request at **Irp->Parameters.StartDevice.AllocatedResources** and **Irp->Parameters.StartDevice.AllocatedResourcesTranslated**, respectively. To connect its interrupt, a driver uses the resources at **AllocatedResourcesTranslated.List.PartialResourceList.PartialDescriptors[]**. The driver must scan the array of partial descriptors for resources of type **CmResourceTypeInterrupt**.

If the driver supplies the storage for the *SpinLock*, it must call **KeInitializeSpinLock** before passing its interrupt spin lock to **IoConnectInterrupt**.

On return from a successful call to **IoConnectInterrupt**, the caller's ISR can be called if interrupts are enabled on the driver's device or if *ShareVector* was set to TRUE.

Callers of **IoConnectInterrupt** must be running at IRQL PASSIVE_LEVEL.

See Also

KeInitializeSpinLock, **KeSynchronizeExecution**, CM_PARTIAL_RESOURCE_DESCRIPTOR

IoCopyCurrentIrpStackLocationToNext

```
VOID
  IoCopyCurrentIrpStackLocationToNext(
  IN PIRP  Irp
  );
```

IoCopyCurrentIrpStackLocationToNext copies the IRP stack parameters from the current I/O stack location to the stack location of the next-lower driver and allows the current driver to set an I/O completion routine.

Parameters

Irp

Points to the IRP.

Include

wdm.h or *ntddk.h*

Comments

A driver calls **IoCopyCurrentIrpStackLocationToNext** to copy the IRP parameters from its stack location to the next-lower driver's stack location.

After calling this routine, a driver typically sets an I/O completion routine with **IoSetCompletionRoutine** before passing the IRP to the next-lower driver with **IoCallDriver**. Drivers that pass on their IRP parameters but do not set an I/O completion routine should call **IoSkipCurrentIrpStackLocation** instead of this routine.

Callers of **IoCopyCurrentIrpStackLocationToNext** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IO_STACK_LOCATION, **IoCallDriver**, **IoSetCompletionRoutine**, **IoSkipCurrentIrpStackLocation**

IoCreateController

```
PCONTROLLER_OBJECT
  IoCreateController(
  IN ULONG  Size
  );
```

IoCreateController allocates memory for and initializes a controller object with a controller extension of a driver-determined size.

Parameters

Size

Specifies the number of bytes to be allocated for the controller extension.

Include

ntddk.h

Return Value

IoCreateController returns a pointer to the controller object or a NULL pointer if memory could not be allocated for the requested device extension.

Comments

A controller object usually represents a physical device controller with attached devices on which a single driver carries out I/O requests. The controller extension is allocated from nonpaged pool and is guaranteed to be accessible by any driver routine and in an arbitrary thread context.

The controller object is used to synchronize I/O operations to target devices for which I/O requests can come in concurrently to a single, monolithic driver. A driver also might use a controller object to synchronize operations through device channels.

If **IoCreateController** returns NULL, the driver should fail device start-up.

Callers of **IoCreateController** must be running at IRQL PASSIVE_LEVEL.

See Also

CONTROLLER_OBJECT, **IoAllocateController**, **IoFreeController**, **IoDeleteController**

IoCreateDevice

```
NTSTATUS
  IoCreateDevice(
  IN PDRIVER_OBJECT  DriverObject,
  IN ULONG  DeviceExtensionSize,
  IN PUNICODE_STRING  DeviceName  OPTIONAL,
  IN DEVICE_TYPE  DeviceType,
  IN ULONG  DeviceCharacteristics,
  IN BOOLEAN  Exclusive,
  OUT PDEVICE_OBJECT  *DeviceObject
  );
```

IoCreateDevice allocates memory for and initializes a device object for use by a driver. A device object represents a physical, virtual, or logical device that the driver is supporting.

Parameters

DriverObject

Points to the driver object for the caller. Each driver receives a pointer to its driver object in a parameter to its **DriverEntry** routine. PnP function and filter drivers also receive a driver object pointer in their AddDevice routines.

DeviceExtensionSize

Specifies the driver-determined number of bytes to be allocated for the device extension of the device object. The internal structure of the device extension is driver-defined. A driver uses the device extension to maintain context about the I/O operations on the device represented by the *DeviceObject*.

DeviceName

Optionally points to a buffer containing a zero-terminated Unicode string that names the device object. The string must be a full path name.

Typically, only Physical Device Objects (PDOs), which are created by PnP bus drivers, are named. PnP function drivers and filter drivers should not specify a *DeviceName* for a Functional Device Object (FDO) or filter device object (filter DO). Naming an FDO or filter DO bypasses the PnP Manager's security. If a user-mode component needs a symbolic link to the device, the function or filter driver should register a device interface (see **IoRegisterDeviceInterface**). If a kernel-mode component needs a legacy device name, the driver must name the FDO, but naming is not recommended.

DeviceType

Specifies one of the system-defined FILE_DEVICE_*XXX* constants indicating the type of device (such as FILE_DEVICE_DISK, FILE_DEVICE_KEYBOARD, etc.) or a driver-

defined value for a new type of device. For more information on device types, see *Determining Required I/O Support by Device Object Type*.

DeviceCharacteristics

Specifies one or more system-defined constants, ORed together, that provide additional information about the driver's device. The constants include:

FILE_AUTOGENERATED_DEVICE_NAME

Directs the I/O Manager to generate a name for the device, instead of the caller specifying a *DeviceName* when calling this routine. The I/O Manager ensures that the name is unique. This characteristic is typically specified by a PnP bus driver to generate a name for a physical device object (PDO) for a child device on its bus. This characteristic is new for Windows 2000 and Windows 98.

FILE_DEVICE_IS_MOUNTED

Indicates that a filesystem is mounted on the device. Drivers should not set this characteristic.

FILE_DEVICE_SECURE_OPEN

(Windows 2000 and Windows NT® SP5 only)

Directs the I/O manager to apply the security descriptor of the device object to relative opens and trailing filename opens on the device.

FILE_FLOPPY_DISKETTE

Indicates that the device is a floppy disk device.

FILE_READ_ONLY_DEVICE

Indicates that the device is not writeable.

FILE_REMOTE_DEVICE

Indicates that the device is remote.

FILE_REMOVABLE_MEDIA

Indicates that the storage device supports removeable media.

Note that this characteristic indicates removeable *media*, not a removeable *device*. For example, drivers for JAZ drive devices should specify this characteristic but drivers for PCMCIA flash disks should not.

FILE_VIRTUAL_VOLUME

Indicates that the volume is virtual. Drivers should not set this characteristic.

FILE_WRITE_ONCE_MEDIA

Indicates that the device supports write-once media.

If none of the device characteristics are relevant to your device, specify zero for this parameter.

Exclusive

Indicates whether the device object represents an *exclusive device*. That is, only one handle at a time can send I/O requests to the corresponding device object. If the underlying device supports overlapped I/O, multiple threads of the same process can send requests through a single handle.

DeviceObject

Points to the newly created device object if the call succeeds.

Include

wdm.h or *ntddk.h*

Return Value

IoCreateDevice can return one of the following NTSTATUS values:

STATUS_SUCCESS
STATUS_INSUFFICIENT_RESOURCES
STATUS_OBJECT_NAME_EXISTS
STATUS_OBJECT_NAME_COLLISION

Comments

IoCreateDevice creates a device object and returns a pointer to the object. The caller is responsible for deleting the object when it is no longer needed by calling **IoDeleteDevice**.

PnP drivers call this routine to create PDOs, FDOs, and filter DOs. See the *Plug and Play, Power Management, and Setup Design Guide* for information about the kinds of PnP drivers and their associated device objects. Legacy, non-PnP drivers call this routine to create legacy device objects.

Be careful to specify the *DeviceType* and *DeviceCharacteristics* values in the correct parameters. Both parameters use system-defined FILE_*XXX* constants and some driver writers specify the values in the wrong parameters by mistake.

If a PnP function or filter driver for a device sets any of the following *DeviceCharacteristics*, the PnP Manager propagates the characteristic(s) to the FDO and filter DOs in the device stack:

FILE_DEVICE_SECURE_OPEN
FILE_FLOPPY_DISKETTE
FILE_READ_ONLY_DEVICE

FILE_REMOVABLE_MEDIA
FILE_WRITE_ONCE_MEDIA

This routine allocates space in nonpaged pool for a driver-defined device extension associated with the device object, so that the device extension is accessible to the driver in any execution context and at any IRQL. The returned device extension is initialized with zeros.

The caller is responsible for setting certain fields in the returned device object, such as the **Flags** field, and for initializing the device extension with any driver-defined information. For other operations required on new device objects, see the *Plug and Play, Power Management, and Setup Design Guide* or the device-type-specific documentation for your device.

Device objects for disks, tapes, CD ROMs, and RAM disks are given a Volume Parameter Block (VPB) that is initialized to indicate that the volume has never been mounted on the device.

If a driver's call to **IoCreateDevice** returns an error, it should release any resources it allocated for that device.

A PnP bus driver calls **IoCreateDevice** when it is enumerating a new device in response to an IRP_MN_QUERY_DEVICE_RELATIONS for **BusRelations**. A PnP function or filter driver calls **IoCreateDevice** in its AddDevice routine.

Callers of **IoCreateDevice** must be running at IRQL PASSIVE_LEVEL.

See Also

DEVICE_OBJECT, **IoAttachDevice**, **IoAttachDeviceToDeviceStack**, **IoCreateSymbolicLink**, **IoDeleteDevice**

IoCreateFile

```
NTSTATUS
    IoCreateFile(
        OUT PHANDLE FileHandle,
        IN ACCESS_MASK DesiredAccess,
        IN POBJECT_ATTRIBUTES ObjectAttributes,
        OUT PIO_STATUS_BLOCK IoStatusBlock,
        IN PLARGE_INTEGER AllocationSize OPTIONAL,
        IN ULONG FileAttributes,
        IN ULONG ShareAccess,
        IN ULONG Disposition,
        IN ULONG CreateOptions,
        IN PVOID EaBuffer OPTIONAL,
        IN ULONG EaLength,
        IN CREATE_FILE_TYPE CreateFileType,
```

```
    IN PVOID ExtraCreateParameters OPTIONAL,
    IN ULONG Options
    ) ;
```

IoCreateFile either causes a new file or directory to be created, or it opens an existing file, device, directory, or volume, giving the caller a handle for the file object. This handle can be used by subsequent calls to manipulate data within the file or the file object's state or attributes.

Parameters

FileHandle

Points to a variable that receives the file handle if the call is successful.

DesiredAccess

Specifies the type of access that the caller requires to the file or directory. The set of system-defined *DesiredAccess* flags determines the following specific access rights for file objects:

DesiredAccess Flags	Meaning
DELETE	The file can be deleted.
FILE_READ_DATA	Data can be read from the file.
FILE_READ_ATTRIBUTES	*FileAttributes* flags, described later, can be read.
FILE_READ_EA	Extended attributes associated with the file can be read. This flag is irrelevant to device and intermediate drivers.
READ_CONTROL	The access control list (ACL) and ownership information associated with the file can be read.
FILE_WRITE_DATA	Data can be written to the file.
FILE_WRITE_ATTRIBUTES	*FileAttributes* flags can be written.
FILE_WRITE_EA	Extended attributes (EAs) associated with the file can be written. This flag is irrelevant to device and intermediate drivers.
FILE_APPEND_DATA	Data can be appended to the file.
WRITE_DAC	The discretionary access control list (DACL) associated with the file can be written.
WRITE_OWNER	Ownership information associated with the file can be written.
SYNCHRONIZE	The returned *FileHandle* can be waited on to synchronize with the completion of an I/O operation.
FILE_EXECUTE	Data can be read into memory from the file using system paging I/O. This flag is irrelevant to device and intermediate drivers.

Callers of **IoCreateFile** can specify one or a combination of the following, possibly ORed with additional compatible flags from the preceding **DesiredAccess Flags** list, for any file object that does not represent a directory file:

DesiredAccess to File Values	Maps to *DesiredAccess* Flags
GENERIC_READ	STANDARD_RIGHTS_READ, FILE_READ_DATA, FILE_READ_ATTRIBUTES, and FILE_READ_EA
GENERIC_WRITE	STANDARD_RIGHTS_WRITE, FILE_WRITE_DATA, FILE_WRITE_ATTRIBUTES, FILE_WRITE_EA, and FILE_APPEND_DATA
GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE, SYNCHRONIZE, and FILE_EXECUTE. This value is irrelevant to device and intermediate drivers.

The STANDARD_RIGHTS_*XXX* are predefined system values used to enforce security on system objects.

To open or create a directory file, as also indicated with the *CreateOptions* parameter, callers of **IoCreateFile** can specify one or a combination of the following, possibly ORed with one or more compatible flags from the preceding **DesiredAccess Flags** list:

DesiredAccess to Directory Values	Meaning
FILE_LIST_DIRECTORY	Files in the directory can be listed.
FILE_TRAVERSE	The directory can be traversed: that is, it can be part of the pathname of a file.

The FILE_READ_DATA, FILE_WRITE_DATA, FILE_EXECUTE, and FILE_APPEND_DATA *DesiredAccess* flags are incompatible with creating or opening a directory file.

ObjectAttributes

Points to a structure already initialized with **InitializeObjectAttributes**. Members of this structure for a file object include the following:

Member	Value
ULONG Length	Specifies the number of bytes of *ObjectAttributes* data supplied. This value must be at least **sizeof**(OBJECT_ATTRIBUTES).
PUNICODE_STRING ObjectName	Points to a buffered Unicode string naming the file to be created or opened. This value must be a fully qualified file specification or the name of a device object, unless it is the name of a file relative to the directory specified by **RootDirectory.** For example, \Device\Floppy1\myfile.dat or \??\B:\myfile.dat could be the fully qualified file specification, provided that the floppy driver and overlying file system are already loaded. (Note: \?? replaces \DosDevices as the name of the Win32 object namespace. \DosDevices will still work, but \?? is translated faster by the object manager.)

Continued

Member	Value
HANDLE RootDirectory	Optionally specifies a handle to a directory obtained by a preceding call to **IoCreateFile**. If this value is NULL, the **ObjectName** member must be a fully qualified file specification that includes the full path to the target file. If this value is nonNULL, the **ObjectName** member specifies a file name relative to this directory.
PSECURITY_DESCRIPTOR SecurityDescriptor	Optionally specifies a security descriptor to be applied to a file. ACLs specified by such a security descriptor are only applied to the file when it is created. If the value is NULL when a file is created, the ACL placed on the file is file-system-dependent; most file systems propagate some part of such an ACL from the parent directory file combined with the caller's default ACL. Device and intermediate drivers can set this member to NULL.
PSECURITY_QUALITY_OF_SERVICE SecurityQualityOfService	Specifies the access rights a server should be given to the client's security context. This value is non-NULL only when a connection to a protected server is established, allowing the caller to control which parts of the caller's security context are made available to the server and whether the server is allowed to impersonate the caller. Device and intermediate drivers usually set this member to NULL.
ULONG Attributes	Is a set of flags that controls the file object attributes. This value can be zero or OBJ_CASE_INSENSITIVE, which indicates that name-lookup code should ignore the case of **ObjectName** rather than performing an exact-match search. The value OBJ_INHERIT is irrelevant to device and intermediate drivers.

IoStatusBlock

Points to a variable that receives the final completion status and information about the requested operation. On return from **IoCreateFile**, the **Information** member contains one of the following values:

FILE_CREATED
FILE_OPENED
FILE_OVERWRITTEN
FILE_SUPERSEDED
FILE_EXISTS
FILE_DOES_NOT_EXIST

AllocationSize

Optionally specifies the initial allocation size in bytes for the file. A nonzero value has no effect unless the file is being created, overwritten, or superseded.

FileAttributes

Explicitly specified attributes are applied only when the file is created, superseded, or, in some cases, overwritten. By default, this value is FILE_ATTRIBUTE_NORMAL, which can be overridden by any other flag or by an ORed combination of compatible flags. Possible *FileAttributes* flags include the following:

FileAttributes Flags	Meaning
FILE_ATTRIBUTE_NORMAL	A file with standard attributes should be created.
FILE_ATTRIBUTE_READONLY	A read-only file should be created.
FILE_ATTRIBUTE_HIDDEN	A hidden file should be created.
FILE_ATTRIBUTE_SYSTEM	A system file should be created.
FILE_ATTRIBUTE_ARCHIVE	The file should be marked so that it will be archived.
FILE_ATTRIBUTE_TEMPORARY	A temporary file should be created.
FILE_ATTRIBUTE_ATOMIC_WRITE	An atomic-write file should be created. This flag is irrelevant to device and intermediate drivers.
FILE_ATTRIBUTE_XACTION_WRITE	A transaction-write file should be created. This flag is irrelevant to device and intermediate drivers.

ShareAccess

Specifies the type of share access that the caller would like to the file, as zero, or as one or a combination of the following:

ShareAccess Flags	Meaning
FILE_SHARE_READ	The file can be opened for read access by other threads' calls to **IoCreateFile**.
FILE_SHARE_WRITE	The file can be opened for write access by other threads' calls to **IoCreateFile**.
FILE_SHARE_DELETE	The file can be opened for delete access by other threads' calls to **IoCreateFile**.

Device and intermediate drivers usually set *ShareAccess* to zero, which gives the caller exclusive access to the open file.

Disposition

Specifies what to do, depending on whether the file already exists, as one of the following:

Disposition Values	Meaning
FILE_SUPERSEDE	If the file already exists, replace it with the given file. If it does not, create the given file.

Continued

Disposition Values	Meaning
FILE_CREATE	If the file already exists, fail the request and do not create or open the given file. If it does not, create the given file.
FILE_OPEN	If the file already exists, open it instead of creating a new file. If it does not, fail the request and do not create a new file.
FILE_OPEN_IF	If the file already exists, open it. If it does not, create the given file.
FILE_OVERWRITE	If the file already exists, open it and overwrite it. If it does not, fail the request.
FILE_OVERWRITE_IF	If the file already exists, open it and overwrite it. If it does not, create the given file.

CreateOptions

Specifies the options to be applied when creating or opening the file, as a compatible combination of the following flags:

CreateOptions Flags	Meaning
FILE_DIRECTORY_FILE	The file being created or opened is a directory file. With this flag, the *Disposition* parameter must be set to one of FILE_CREATE, FILE_OPEN, or FILE_OPEN_IF. With this flag, other compatible *CreateOptions* flags include only the following: FILE_SYNCHRONOUS_IO_ALERT, FILE_SYNCHRONOUS_IO_NONALERT, FILE_WRITE_THROUGH, FILE_OPEN_FOR_BACKUP_INTENT, and FILE_OPEN_BY_FILE_ID.
FILE_NON_DIRECTORY_FILE	The file being opened must not be a directory file or this call will fail. The file object being opened can represent a data file, a logical, virtual, or physical device, or a volume.
FILE_WRITE_THROUGH	System services, FSDs, and drivers that write data to the file must actually transfer the data into the file before any requested write operation is considered complete. This flag is automatically set if the *CreateOptions* flag FILE_NO_INTERMEDIATE_BUFFERING is set.
FILE_SEQUENTIAL_ONLY	All accesses to the file will be sequential.
FILE_RANDOM_ACCESS	Accesses to the file can be random, so no sequential read-ahead operations should be performed on the file by FSDs or the system.

CreateOptions Flags	Meaning
FILE_NO_INTERMEDIATE_BUFFERING	The file cannot be cached or buffered in a driver's internal buffers. This flag is incompatible with the *DesiredAccess* FILE_APPEND_DATA flag.
FILE_SYNCHRONOUS_IO_ALERT	All operations on the file are performed synchronously. Any wait on behalf of the caller is subject to premature termination from alerts. This flag also causes the I/O system to maintain the file position context. If this flag is set, the *DesiredAccess* SYNCHRONIZE flag also must be set.
FILE_SYNCHRONOUS_IO_NONALERT	All operations on the file are performed synchronously. Waits in the system to synchronize I/O queueing and completion are not subject to alerts. This flag also causes the I/O system to maintain the file position context. If this flag is set, the *DesiredAccess* SYNCHRONIZE flag also must be set.
FILE_CREATE_TREE_CONNECTION	Create a tree connection for this file in order to open it over the network. This flag is irrelevant to device and intermediate drivers.
FILE_COMPLETE_IF_OPLOCKED	Complete this operation immediately with an alternate success code if the target file is oplocked, rather than blocking the caller's thread. If the file is oplocked, another caller already has access to the file over the network. This flag is irrelevant to device and intermediate drivers.
FILE_NO_EA_KNOWLEDGE	If the extended attributes on an existing file being opened indicate that the caller must understand EAs to properly interpret the file, fail this request because the caller does not understand how to deal with EAs. Device and intermediate drivers can ignore this flag.
FILE_DELETE_ON_CLOSE	Delete the file when the last handle to it is passed to **ZwClose**.
FILE_OPEN_BY_FILE_ID	The file name contains the name of a device and a 64-bit ID to be used to open the file. This flag is irrelevant to device and intermediate drivers.
FILE_OPEN_FOR_BACKUP_INTENT	The file is being opened for backup intent, hence, the system should check for certain access rights and grant the caller the appropriate accesses to the file before checking the input *DesiredAccess* against the file's security descriptor. This flag is irrelevant to device and intermediate drivers.

EaBuffer

For device and intermediate drivers, this parameter must be a NULL pointer.

EaLength

For device and intermediate drivers, this parameter must be zero.

CreateFileType

Drivers must set this parameter to CreateFileTypeNone.

ExtraCreateParameters

Drivers must set this parameter to NULL.

Options

Specifies options to be used during the creation of the create request. These options can be from the following list:

Options Flags	Meaning
IO_NO_PARAMETER_CHECKING	Indicates that the parameters for this call should not be validated before attempting to issue the create request. Driver writers should use this flag with caution as certain invalid parameters can cause a system failure.
IO_FORCE_ACCESS_CHECK	Indicates that the I/O Manager must check the operation against the file's security descriptor.

Include

wdm.h or *ntddk.h*

Return Value

IoCreateFile either returns STATUS_SUCCESS or an appropriate error status. If it returns an error status, the caller can find additional information about the cause of the failure by checking the *IoStatusBlock*.

Comments

There are two alternate ways to specify the name of the file to be created or opened with **IoCreateFile**:

1. As a fully qualified pathname, supplied in the **ObjectName** member of the input *Object-Attributes*
2. As pathname relative to the directory file represented by the handle in the **RootDirectory** member of the input *ObjectAttributes*

Certain *DesiredAccess* flags and combinations of flags have the following effects:

- For a caller to synchronize an I/O completion by waiting on the returned *FileHandle*, the SYNCHRONIZE flag must be set. Otherwise, a caller that is a device or intermediate driver must synchronize an I/O completion by using an event object.
- If only the FILE_APPEND_DATA and SYNCHRONIZE flags are set, the caller can write only to the end of the file, and any offset information on writes to the file is ignored. However, the file will automatically be extended as necessary for this type of write operation.
- Setting the FILE_WRITE_DATA flag for a file also allows writes beyond the end of the file to occur. The file is automatically extended for this type of write, as well.
- If only the FILE_EXECUTE and SYNCHRONIZE flags are set, the caller cannot directly read or write any data in the file using the returned *FileHandle*: that is, all operations on the file occur through the system pager in response to instruction and data accesses. Device and intermediate drivers should not set the FILE_EXECUTE flag in *DesiredAccess*.

The *ShareAccess* parameter determines whether separate threads can access the same file, possibly simultaneously. Provided that both file openers have the privilege to access a file in the specified manner, the file can be successfully opened and shared. If the original caller of **IoCreateFile** does not specify FILE_SHARE_READ, FILE_SHARE_WRITE, or FILE_SHARE_DELETE, no other open operations can be performed on the file: that is, the original caller is given exclusive access to the file.

In order for a shared file to be successfully opened, the requested *DesiredAccess* to the file must be compatible with both the *DesiredAccess* and *ShareAccess* specifications of all preceding opens that have not yet been released with **ZwClose**. That is, the *DesiredAccess* specified to **IoCreateFile** for a given file must not conflict with the accesses that other openers of the file have disallowed.

The *Disposition* value FILE_SUPERSEDE requires that the caller have DELETE access to a existing file object. If so, a successful call to **IoCreateFile** with FILE_SUPERSEDE on an existing file effectively deletes that file, and then recreates it. This implies that, if the file has already been opened by another thread, it opened the file by specifying a *ShareAccess* parameter with the FILE_SHARE_DELETE flag set. Note that this type of disposition is consistent with the POSIX style of overwriting files.

The *Disposition* values FILE_OVERWRITE_IF and FILE_SUPERSEDE are similar. If **IoCreateFile** is called with a existing file and either of these *Disposition* values, the file will be replaced.

Overwriting a file is semantically equivalent to a supersede operation, except for the following:

- The caller must have write access to the file, rather than delete access. This implies that, if the file has already been opened by another thread, it opened the file with the FILE_SHARE_WRITE flag set in the input *ShareAccess*.
- The specified file attributes are logically ORed with those already on the file. This implies that, if the file has already been opened by another thread, a subsequent caller of **IoCreateFile** cannot disable existing *FileAttributes* flags but can enable additional flags for the same file.

The *CreateOptions* FILE_DIRECTORY_FILE value specifies that the file to be created or opened is a directory file. When a directory file is created, the file system creates an appropriate structure on the disk to represent an empty directory for that particular file system's on-disk structure. If this option was specified and the given file to be opened is not a directory file, or if the caller specified an inconsistent *CreateOptions* or *Dispostion* value, the call to **IoCreateFile** will fail.

The *CreateOptions* FILE_NO_INTERMEDIATE_BUFFERING flag prevents the file system from performing any intermediate buffering on behalf of the caller. Specifying this value places certain restrictions on the caller's parameters to the **Zw..File** routines, including the following:

- Any optional *ByteOffset* passed to **ZwReadFile** or **ZwWriteFile** must be an integral of the sector size.
- The *Length* passed to **ZwReadFile** or **ZwWriteFile**, must be an integral of the sector size. Note that specifying a read operation to a buffer whose length is exactly the sector size might result in a lesser number of significant bytes being transferred to that buffer if the end of the file was reached during the transfer.
- Buffers must be aligned in accordance with the alignment requirement of the underlying device. This information can be obtained by calling **IoCreateFile** to get a handle for the file object that represents the physical device, and, then, calling **ZwQueryInformationFile** with that handle. For a list of the system FILE_XXX_ALIGNMENT values, see *DEVICE_OBJECT* in Chapter 12.
- Calls to **ZwSetInformationFile** with the *FileInformationClass* parameter set to **FilePositionInformation** must specify an offset that is an integral of the sector size.

The *CreateOptions* FILE_SYNCHRONOUS_IO_ALERT and FILE_SYNCHRONOUS_IO_NONALERT, which are mutually exclusive as their names suggest, specify that all I/O operations on the file are to be synchronous as long as they occur through the file object referred to by the returned *FileHandle*. All I/O on such a file is serialized across all threads

using the returned handle. With either of these *CreateOptions*, the *DesiredAccess* SYNCHRONIZE flag must be set so that the I/O Manager will use the file object as a synchronization object. With either of these *CreateOptions* set, the I/O Manager maintains the "file position context" for the file object, an internal, current file position offset. This offset can be used in calls to **ZwReadFile** and **ZwWriteFile**. Its position also can be queried or set with **ZwQueryInformationFile** and **ZwSetInformationFile**.

Callers of **IoCreateFile** must be running at IRQL PASSIVE_LEVEL.

See Also

ZwCreateFile

IoCreateNotificationEvent

```
PKEVENT
  IoCreateNotificationEvent(
  IN PUNICODE_STRING  EventName,
  OUT PHANDLE  EventHandle
  );
```

IoCreateNotificationEvent creates or opens a named notification event used to notify one or more threads of execution that an event has occurred.

Parameters

EventName

Points to a buffer containing a zero-terminated Unicode string that names the event.

EventHandle

Points to a location in which to return a handle for the event object. The handle includes bookkeeping information, such as a reference count and security context.

Include

wdm.h or *ntddk.h*

Return Value

IoCreateNotificationEvent returns a pointer to the created or opened event object or NULL if the event object could not be created or opened.

Comments

IoCreateNotificationEvent creates and opens the event object if it does not already exist. **IoCreateNotificationEvent** sets the state of a new notification event to Signaled. If the event object already exists, **IoCreateNotificationEvent** just opens the event object.

When a notification event is set to the Signaled state it remains in that state until it is explicitly cleared.

Notification events, like synchronization events, are used to coordinate execution. Unlike a synchronization event, a notification event is not auto-resetting. Once a notification event is in the Signaled state, it remains in that state until it is explicitly reset (with a call to **KeClearEvent** or **KeResetEvent**).

To synchronize on a notification event:

1. Open the notification event with **IoCreateNotificationEvent**. Identify the event with the *EventName* string.
2. Wait for the event to be signaled by calling **KeWaitForSingleObject** with the PKEVENT returned by **IoCreateNotificationEvent**. More than one thread of execution can wait on a given notification event. To poll instead of stall, specify a *Timeout* of zero to **KeWaitForSingleObject**.
3. Close the handle to the notification event with **ZwClose** when access to the event is no longer needed.

Callers of **IoCreateNotificationEvent** must be running at IRQL PASSIVE_LEVEL.

See Also

IoCreateSynchronizationEvent, **KeClearEvent**, **KeResetEvent**, **KeSetEvent**, **KeWaitForSingleObject**, **RtlInitUnicodeString**, **ZwClose**

IoCreateSymbolicLink

```
NTSTATUS
  IoCreateSymbolicLink(
  IN PUNICODE_STRING  SymbolicLinkName,
  IN PUNICODE_STRING  DeviceName
  );
```

IoCreateSymbolicLink sets up a symbolic link between a device object name and a user-visible name for the device.

Parameters

SymbolicLinkName

Points to a buffered Unicode string that is the user-visible name.

DeviceName

Points to a buffered Unicode string that is the name of the driver-created device object.

Include

wdm.h or *ntddk.h*

Return Value

IoCreateSymbolicLink returns STATUS_SUCCESS if the symbolic link object was created.

Comments

PnP drivers do not name device objects and therefore should not use this routine. Instead, a PnP driver should call **IoRegisterDeviceInterface** to set up a symbolic link.

Callers of **IoCreateSymbolicLink** must be running at IRQL PASSIVE_LEVEL.

See Also

IoRegisterDeviceInterface, **IoAssignArcName**, **IoCreateUnprotectedSymbolicLink**, **IoDeleteSymbolicLink**

IoCreateSynchronizationEvent

```
PKEVENT
  IoCreateSynchronizationEvent(
  IN PUNICODE_STRING  EventName,
  OUT PHANDLE  EventHandle
  );
```

IoCreateSynchronizationEvent creates or opens a named synchronization event for use in serialization of access to hardware between two otherwise unrelated drivers.

Parameters

EventName

Points to a buffer containing a zero-terminated Unicode string that names the event.

EventHandle

Points to a location in which to return a handle for the event object.

Include

ntddk.h

Return Value

IoCreateSynchronizationEvent returns a pointer to the created or opened event object or NULL if the event object could not be created or opened.

Comments

The event object is created if it does not already exist. **IoCreateSynchronizationEvent** sets the state of a new synchronization event to Signaled. If the event object already exists, it is simply opened. The pair of drivers that use a synchronization event call **KeWaitForSingleObject** with the PKEVENT pointer returned by this routine.

When a synchronization event is set to the Signaled state, a single thread of execution that was waiting on the event is released, and the event is automatically reset to the Not-Signaled state.

To release the event, a driver calls **ZwClose** with the event handle.

Callers of **IoCreateSynchronizationEvent** must be running at IRQL PASSIVE_LEVEL.

See Also

IoCreateNotificationEvent, **KeWaitForSingleObject**, **RtlInitUnicodeString**, **ZwClose**

IoCreateUnprotectedSymbolicLink

```
NTSTATUS
  IoCreateUnprotectedSymbolicLink(
  IN PUNICODE_STRING  SymbolicLinkName,
  IN PUNICODE_STRING  DeviceName
  );
```

IoCreateUnprotectedSymbolicLink sets up an unprotected symbolic link between a device object name and a corresponding Win32®-visible name.

Parameters

SymbolicLinkName

Supplies the symbolic link name as a Unicode string.

DeviceName

Supplies the name of the device object to which the symbolic link name refers.

Include

wdm.h or *ntddk.h*

Return Value

IoCreateUnprotectedSymbolicLink returns the final status of the operation.

Comments

PnP drivers do not name device objects and therefore should not use this routine. Instead, a PnP driver should call **IoRegisterDeviceInterface** to set up a symbolic link.

IoCreateUnprotectedSymbolicLink can be used by drivers if the user needs to be able to manipulate the symbolic link. For example, the parallel and serial drivers create unprotected symbolic links for LPTx and COMx, so that users can manipulate and reassign them by using the MODE command.

In general, drivers should call this routine instead of **IoCreateSymbolicLink** if a protected subsystem lets end users change what a named device references as, for example, when using LPT1 to access a network printer.

Callers of **IoCreateUnprotectedSymbolicLink** must be running at IRQL PASSIVE_LEVEL.

See Also

IoRegisterDeviceInterface, **IoAssignArcName**, **IoCreateSymbolicLink**, **IoDeleteSymbolicLink**

IoDeassignArcName

```
VOID
  IoDeassignArcName(
  IN PUNICODE_STRING  ArcName
  );
```

IoDeassignArcName removes a symbolic link between the ARC name for a device and the named device object. This is generally called if the driver is deleting the device object, for example, when the driver is unloading.

Parameters

ArcName

Points to a buffered Unicode string that is the ARC name.

Include

ntddk.h

Comments

Callers of **IoDeassignArcName** must be running at IRQL PASSIVE_LEVEL.

See Also

IoAssignArcName

IoDeleteController

```
VOID
  IoDeleteController(
  IN PCONTROLLER_OBJECT  ControllerObject
  );
```

IoDeleteController removes a given controller object from the system, for example, when the driver that created it is being unloaded.

Parameters

ControllerObject

Points to the controller object to be released.

Include

ntddk.h

Comments

IoDeleteController deallocates the memory for the controller object, including the controller extension.

This routine must be called when a driver that created a controller object is being unloaded or when the driver encounters a fatal error during device start-up, such as being unable to properly initialize a physical device.

A driver must release certain resources for which the driver supplied storage in its controller extension before it calls **IoDeleteController**. For example, if the driver stores the pointer to its interrupt object(s) in the controller extension, it must call **IoDisconnectInterrupt** before **IoDeleteController**.

Callers of **IoDeleteController** must be running at IRQL PASSIVE_LEVEL.

See Also

IoCreateController, IoDisconnectInterrupt

IoDeleteDevice

```
VOID
  IoDeleteDevice(
  IN PDEVICE_OBJECT  DeviceObject
  );
```

IoDeleteDevice removes a device object from the system, for example, when the underlying device is removed from the system.

Parameters

DeviceObject

Points to the device object to be deleted.

Include

wdm.h or *ntddk.h*

Comments

When handling a PnP IRP_MN_REMOVE_DEVICE request, a PnP driver calls **IoDeleteDevice** to delete any associated device objects.

A legacy driver should call this routine when it is being unloaded or when its **DriverEntry** routine encounters a fatal initialization error, such as being unable to properly initialize a physical device. This routine also is called when a driver reconfigures its devices dynamically. For example, a disk driver called to repartition a disk would call **IoDeleteDevice** to tear down the device objects representing partitions to be replaced.

A driver must release certain resources for which the driver supplied storage in its device extension before it calls **IoDeleteDevice**. For example, if the driver stores the pointer to its interrupt object(s) in the device extension, it must call **IoDisconnectInterrupt** before calling **IoDeleteDevice**.

A driver can call **IoDeleteDevice** only once for a given device object.

When a driver calls **IoDeleteDevice**, the I/O Manager deletes the target device object if there are no outstanding references to it. However, if any outstanding references remain, the I/O Manager marks the device object as "delete pending" and deletes the device object when the references are released.

Callers of **IoDeleteDevice** must be running at IRQL PASSIVE_LEVEL.

See Also

IoCreateDevice, IoDisconnectInterrupt

IoDeleteSymbolicLink

```
NTSTATUS
  IoDeleteSymbolicLink(
  IN PUNICODE_STRING  SymbolicLinkName
  );
```

IoDeleteSymbolicLink removes a symbolic link from the system.

Parameters

SymbolicLinkName

Points to a buffered Unicode string that is the user-visible name for the symbolic link.

Include

wdm.h or *ntddk.h*

Return Value

IoDeleteSymbolicLink returns STATUS_SUCCESS if the symbolic link object is deleted.

Comments

Callers of **IoDeleteSymbolicLink** must be running at IRQL PASSIVE_LEVEL.

See Also

IoCreateSymbolicLink, **IoCreateUnprotectedSymbolicLink**, **IoDeassignArcName**

IoFreeAdapterChannel

```
VOID
  IoFreeAdapterChannel(
  IN PADAPTER_OBJECT  AdapterObject
  );
```

IoFreeAdapterChannel is obsolete and exported only to support existing drivers. See **FreeAdapterChannel** instead.

IoFreeController

```
VOID
  IoFreeController(
  IN PCONTROLLER_OBJECT  ControllerObject
  );
```

IoFreeController releases a previously allocated controller object when the driver has completed an I/O request.

Parameters

ControllerObject

Points to the driver's controller object, which was allocated for the current I/O operation on a particular device by calling **IoAllocateController**.

Include

ntddk.h

Comments

The connection between the current target device object and the controller object is released only if no requests are currently queued to the same device. Otherwise, the driver's ControllerControl routine is called with the next IRP bound through the device controller to the target device.

Callers of **IoFreeController** must be running at IRQL DISPATCH_LEVEL.

See Also

IoAllocateController, **IoCreateController**, **IoDeleteController**

IoFreeIrp

```
VOID
  IoFreeIrp(
  IN PIRP  Irp
  );
```

IoFreeIrp releases a caller-allocated IRP from the caller's IoCompletion routine.

Parameters

Irp

Points to the IRP that is to be released.

Include

wdm.h or *ntddk.h*

Comments

This routine is the reciprocal to **IoAllocateIrp** or **IoBuildAsynchronousFsdRequest**. The released IRP must have been allocated by the caller.

This routine also releases an IRP allocated with **IoMakeAssociatedIrp** in which the caller set up its IoCompletion routine that returns STATUS_MORE_PROCESSING_REQUIRED for the associated IRP.

Callers of **IoFreeIrp** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateIrp, **IoBuildAsynchronousFsdRequest**, **IoMakeAssociatedIrp**, **IoSetCompletionRoutine**

IoFreeMapRegisters

```
VOID
  IoFreeMapRegisters(
  IN PADAPTER_OBJECT  AdapterObject,
  IN PVOID  MapRegisterBase,
  IN ULONG  NumberOfMapRegisters
  );
```

IoFreeMapRegisters is obsolete and exported only to support existing drivers. See **FreeMapRegisters** instead.

IoFreeMdl

```
VOID
  IoFreeMdl(
  IN PMDL  Mdl
  );
```

IoFreeMdl releases a caller-allocated MDL.

Parameters

Mdl

Points to the MDL to be released.

Include

wdm.h or *ntddk.h*

Comments

If a driver allocates an MDL to describe a buffer, it must explicitly release the MDL when operations on the buffer are done.

Callers of **IoFreeMdl** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateMdl, **IoBuildPartialMdl**

IoFreeWorkItem

```
VOID
  IoFreeWorkItem(
  IN PIO_WORKITEM pIOWorkItem
  );
```

IoFreeWorkItem frees the specified work item.

Parameters

pIOWorkItem

Pointer to a private IO_WORKITEM structure that was returned by a previous call to **IoAllocateWorkItem**.

Include

wdm.h or *ntddk.h*

Comments

Drivers should not make any assumptions about the format of the IO_WORKITEM structure nor should they attempt to access information that is contained in this private structure.

See Also

IoAllocateWorkItem, **IoQueueWorkItem**

IoGetAttachedDeviceReference

```
PDEVICE_OBJECT
  IoGetAttachedDeviceReference(
  IN PDEVICE_OBJECT  DeviceObject
  );
```

IoGetAttachedDeviceReference returns a pointer to the highest level device object in a driver stack and increments the reference count on that object.

Parameters

DeviceObject

Points to the device object for which the topmost attached device object is retrieved.

Include

wdm.h or *ntddk.h*

Return Value

IoGetAttachedDeviceReference returns a pointer to the highest level device object in a stack of attached device objects after incrementing the reference count on the object.

Comments

If the device object at *DeviceObject* has no device objects attached to it, *DeviceObject* and the returned pointer are equal.

Device driver writers must ensure that when they have completed all operations that required them to make this call, that they call **ObDereferenceObject** with the device object pointer returned by this routine. Failure to do so will prevent the system from freeing or deleting the device object because of an outstanding reference count.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

ObDereferenceObject

IoGetBootDiskInformation

```
NTSTATUS
IoGetBootDiskInformation(
  IN OUT PBOOTDISK_INFORMATION BootDiskInformation,
  IN ULONG Size
  );
```

IoGetBootDiskInformation returns the offset and signature of the boot disk and the system disk.

Parameters

BootDiskInformation

Pointer to a caller-allocated buffer that is used to output a BOOTDISK_INFORMATION structure:

```
typedef struct _BOOTDISK_INFORMATION {
    LONGLONG BootPartitionOffset;
    LONGLONG SystemPartitionOffset;
    ULONG BootDeviceSignature;
    ULONG SystemDeviceSignature;
} BOOTDISK_INFORMATION, *PBOOTDISK_INFORMATION;
```

Size

Specifies the size in bytes of a BOOTDISK_INFORMATION structure.

Include

ntddk.h

Return Value

IoGetBootDiskInformation returns one of the following status values:

STATUS_SUCCESS

STATUS_TOO_LATE

The Loader Block has already been freed.

STATUS_INVALID_PARAMETER

The value of *Size* is less than the size in bytes of a BOOTDISK_INFORMATION structure.

Comments

IoGetBootDiskInformation can be used only by boot drivers that have registered for a callback after disk devices have started.

IoGetConfigurationInformation

```
PCONFIGURATION_INFORMATION
  IoGetConfigurationInformation(
  VOID
  );
```

IoGetConfigurationInformation returns a pointer to the I/O Manager's global configuration information structure, which contains the current values for how many physical disk, floppy, CD-ROM, tape, SCSI HBA, serial, and parallel devices have device objects created to represent them by drivers as they are loaded.

Include

ntddk.h

Return Value

IoGetConfigurationInformation returns a pointer to the configuration information structure. This structure is defined as follows:

```
typedef struct _CONFIGURATIONAL_INFORMATION{
    //
    // Each field indicates the total number of physical
    // devices of a particular type in the machine. The value
    // should be used by the driver to determine the digit
    // suffix for device object names. This field must be
    // updated as the driver finds new devices of its own.
    //
    ULONG DiskCount;            // Count of hard disks so far.
    ULONG FloppyCount;          // Count of floppy drives so far.
    ULONG CDRomCount;           // Count of CD-ROM drives so far.
    ULONG TapeCount;            // Count of tape drives so far.
    ULONG ScsiPortCount;        // Count of HBAs so far.
    ULONG SerialCount;          // Count of serial ports so far.
    ULONG ParallelCount;        // Count of parallel ports so far.
    //
    // The next two fields indicate ownership of
    // either of the two I/O address spaces
    // that are used by WD1003-compatible disk controllers.
    //
    BOOLEAN AtDiskPrimaryAddressClaimed;       //0x1F0-0x1FF
    BOOLEAN AtDiskSecondaryAddressClaimed;     //0x170-0x17F
} CONFIGURATION_INFORMATION,*PCONFIGURATION_INFORMATION
```

Comments

Certain types of device drivers can use the configuration information structure's values to construct device object names with appropriate digit suffixes when each driver creates its device objects. Note that the digit suffix for device object names is a zero-based count, while the counts maintained in the configuration information structure represent the number of device objects of a particular type already created. That is, the configuration information counts are one-based.

Any driver that calls **IoGetConfigurationInformation** must increment the count for its kind of device in this structure when it creates a device object to represent a physical device.

The system-supplied SCSI port driver supplies the count of SCSI HBAs present in the computer. SCSI class drivers can read this value to determine how many HBA-specific miniport drivers might control a SCSI bus with an attached device of the class driver's type.

The configuration information structure also contains a value indicating whether an already loaded driver has claimed either of the "AT" disk I/O address ranges.

Callers of **IoGetConfigurationInformation** must be running at IRQL PASSIVE_LEVEL.

See Also

HalAssignSlotResources, HalGetBusData, HalGetBusDataByOffset, IoAssignResources, IoQueryDeviceDescription, IoReportResourceUsage

IoGetCurrentIrpStackLocation

```
PIO_STACK_LOCATION
  IoGetCurrentIrpStackLocation(
  IN PIRP  Irp
  );
```

IoGetCurrentIrpStackLocation returns a pointer to the caller's stack location in the given IRP.

Parameters

Irp

Points to the IRP.

Include

wdm.h or *ntddk.h*

Return Value

The routine returns a pointer to the I/O stack location for the driver.

Comments

Every driver must call **IoGetCurrentIrpStackLocation** with each IRP it is sent to get any parameters for the current request. Unless a driver supplies a Dispatch routine for each IRP_MJ_*XXX* that driver handles, the driver also must check its I/O stack location in the IRP to determine what operation is being requested.

Intermediate and highest-level drivers also call **IoGetCurrentIrpStackLocation** so that they can copy pertinent data from their own stack location into that of the next-lower driver whenever they pass a request on to lower drivers.

See Also

IO_STACK_LOCATION, **IoCallDriver**, **IoGetNextIrpStackLocation**, **IoSetNextIrpStackLocation**

IoGetCurrentProcess

```
PEPROCESS
  IoGetCurrentProcess( );
```

Include

wdm.h or *ntddk.h*

Return Value

IoGetCurrentProcess returns a pointer to the current process.

Comments

In general, highest-level drivers, particularly file systems, are most likely to call this routine. An intermediate or underlying device driver seldom is called in the context of a thread that originates the current I/O request that the driver is processing, so it cannot get access to such a thread's process space.

Callers of **IoGetCurrentProcess** must be running at IRQL PASSIVE_LEVEL.

See Also

PsGetCurrentThread

IoGetDeviceInterfaceAlias

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see **IoGetDeviceInterfaceAlias** in that book for a full reference.

IoGetDeviceInterfaces

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see **IoGetDeviceInterfaces** in that book for a full reference.

IoGetDeviceObjectPointer

```
NTSTATUS
  IoGetDeviceObjectPointer(
  IN PUNICODE_STRING  ObjectName,
  IN ACCESS_MASK  DesiredAccess,
  OUT PFILE_OBJECT  *FileObject,
  OUT PDEVICE_OBJECT  *DeviceObject
  );
```

IoGetDeviceObjectPointer returns a pointer to a named device object and corresponding file object if the requested access to the objects can be granted.

Parameters

ObjectName

Points to a buffer containing a Unicode string that is the name of the device object.

DesiredAccess

Specifies one or more (ORed) system-defined constants, usually FILE_READ_DATA, (infrequently) FILE_WRITE_DATA, and/or FILE_ALL_ACCESS, requesting access rights to the object.

FileObject

Points to the file object that represents the corresponding device object to user-mode code if the call is successful.

DeviceObject

Points to the device object that represents the named logical, virtual, or physical device if the call is successful.

Include

wdm.h or *ntddk.h*

Return Value

IoGetDeviceObjectPointer can return one of the following NTSTATUS values:

STATUS_SUCCESS
STATUS_OBJECT_TYPE_MISMATCH
STATUS_INVALID_PARAMETER
STATUS_PRIVILEGE_NOT_HELD
STATUS_INSUFFICIENT_RESOURCES
STATUS_OBJECT_NAME_INVALID

Comments

IoGetDeviceObjectPointer establishes a "connection" between the caller and the next-lower-level driver. A successful caller can use the returned device object pointer to initialize its own device object(s). It can also be used as as an argument to **IoAttachDeviceToDeviceStack**, **IoCallDriver**, and any routine that creates IRPs for lower drivers. The returned pointer is a required argument to **IoCallDriver**.

This routine also returns a pointer to the corresponding file object. When unloading, a driver can dereference the file object as a means of indirectly dereferencing the device object. To do so, the driver calls **ObDereferenceObject** from its Unload routine, passing the file object pointer returned by **GetDeviceObjectPointer**. Failure to dereference the device object in a driver's Unload routine prevents the next-lower driver from being unloaded. However, drivers that close the file object before the unload process must take out an extra reference on the device object before dereferencing the file object. Otherwise, dereferencing the file object can lead to a premature deletion of the device object.

To get a pointer to the highest-level driver in the file system driver stack, a driver must ensure that the file system is mounted; if it is not, this routine traverses the storage device stack. To ensure that the file system is mounted on the storage device, the driver must specify an appropriate access mask, such as FILE_READ_DATA or FILE_WRITE_ATTRIBUTES, in the *DesiredAccess* parameter. Specifying FILE_READ_ATTRIBUTES does not cause the file system to be mounted.

After any higher-level driver has chained itself over another driver by successfully calling this routine, the higher-level driver must set the **StackSize** field in its device object to that of the next-lower-level driver's device object plus one.

Callers of **IoGetDeviceObjectPointer** must be running at IRQL PASSIVE_LEVEL.

See Also

DEVICE_OBJECT, **IoAllocateIrp**, **IoAttachDevice**, **IoAttachDeviceToDeviceStack**, **ObDereferenceObject**, **ObReferenceObjectByPointer**

IoGetDeviceProperty

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoGetDeviceProperty*** in that book for a full reference.

IoGetDeviceToVerify

```
PDEVICE_OBJECT
  IoGetDeviceToVerify(
  IN PETHREAD Thread
  );
```

IoGetDeviceToVerify returns a pointer to the device object, representing a removable-media device, that is the target of the given thread's I/O request.

Parameters

Thread

Points to the thread for which a highest-level driver is attempting to verify the validity of the media on which the thread has opened a file.

Include

ntddk.h

Return Value

IoGetDeviceToVerify returns a pointer to the device object representing a device on which the media should be verified, or it returns NULL.

Comments

In general, highest-level drivers, particularly file systems, are most likely to call this routine.

An underlying removable-media device driver is responsible for notifying higher-level drivers, particularly the file system, when the media appears to have changed since the last access to the target device. For more information about handling removable media, see the *Kernel-Mode Drivers Design Guide*.

Callers of **IoGetDeviceToVerify** must be running at IRQL PASSIVE_LEVEL.

See Also

IoIsErrorUserInduced, **IoSetHardErrorOrVerifyDevice**, **PsGetCurrentThread**

IoGetDmaAdapter

```
PDMA_ADAPTER
  IoGetDmaAdapter(
  IN PDEVICE_OBJECT  PhysicalDeviceObject,
  IN PDEVICE_DESCRIPTION  DeviceDescription,
  IN OUT PULONG  NumberOfMapRegisters
  );
```

IoGetDmaAdapter returns a pointer to the DMA adapter structure for a physical device object.

Parameters

PhysicalDeviceObject

Points to the physical device object for the device requesting the DMA adapter structure.

DeviceDescription

Points to a DEVICE_DESCRIPTION structure, which describes the attributes of the physical device.

NumberOfMapRegisters

Points to, on output, the maximum number of map registers that the driver can allocate for any DMA transfer operation.

Include

wdm.h or *ntddk.h*

Return Value

IoGetDmaAdapter returns a pointer to a DMA adapter structure that contains function pointers to the system-defined set of DMA operations. If an adapter structure cannot be allocated, the routine returns NULL.

Comments

Before calling this routine, a driver must zero-initialize the structure passed at *Device-Description* and then supply the relevant information for its device.

When **IoGetDmaAdapter** returns a valid pointer, a driver can use the pointers to functions within the DMA_ADAPTER structure to perform subsequent DMA operations.

PnP drivers call **IoGetDmaAdapter** when handling a PnP IRP_MN_START_DEVICE request for a device. This IRP includes information about the device's hardware resources that the driver must supply in the *DeviceDescription* structure.

In *NumberOfMapRegisters*, the caller specifies the optimal number of map registers it can use. On output, the I/O Manager returns the number of map registers it allocated. Drivers should check the returned value; there is no guarantee a driver will receive the same number of map registers it requested.

To free the adapter object, the driver should call **PutDmaAdapter** through the pointer returned in the DMA_ADAPTER structure.

Drivers must call this routine while running at IRQL PASSIVE_LEVEL.

See Also

DEVICE_DESCRIPTION, DMA_ADAPTER, **PutDmaAdapter**

IoGetDriverObjectExtension

```
PVOID
  IoGetDriverObjectExtension(
  IN PDRIVER_OBJECT  DriverObject,
  IN PVOID  ClientIdentificationAddress
  );
```

IoGetDriverObjectExtension retrieves a previously allocated per-driver context area.

Parameters

DriverObject

Specifies the driver object with which the context area is associated.

ClientIdentificationAddress

Specifies the unique identifier, provided when it was allocated, of the context area to be retrieved.

Include

wdm.h or *ntddk.h*

Return Value

IoGetDriverObjectExtension returns a pointer to the context area, if any or returns NULL.

Comments

Drivers call **IoGetDriverObjectExtension** to retrieve a pointer to a previously allocated extension area.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateDriverObjectExtension

IoGetFileObjectGenericMapping

```
PGENERIC_MAPPING
  IoGetFileObjectGenericMapping( );
```

IoGetFileObjectGenericMapping returns information about the mapping between each generic access right and the set of specific access rights for file objects.

Include

ntddk.h

Return Value

IoGetFileObjectGenericMapping returns a pointer to the generic mapping for file objects.

Comments

The generic mapping structure is defined as follows:

```
typedef struct _GENERIC_MAPPING {
    ACCESS_MASK GenericRead;
    ACCESS_MASK GenericWrite;
    ACCESS_MASK GenericExecute;
    ACCESS_MASK GenericAll;
} GENERIC_MAPPING;

typedef GENERIC_MAPPING *PGENERIC_MAPPING;
```

Callers of **IoGetFileObjectGenericMapping** must be running at IRQL PASSIVE_LEVEL.

See Also

IoCheckShareAccess, IoSetShareAccess, ZwCreateFile

IoGetFunctionCodeFromCtlCode

```
ULONG
  IoGetFunctionCodeFromCtlCode(
  IN ULONG ControlCode
  );
```

IoGetFunctionCodeFromCtlCode returns the value of the *Function* in a given IOCTL_*XXX* control code.

Parameters

ControlCode

Points to the IOCTL *_XXX* (or FSCTL_*XXX*) in the driver's I/O stack location of the IRP at **Parameters.DeviceIoControl.IoControlCode**.

Include

wdm.h or *ntddk.h*

Return Value

IoGetFunctionCodeFromCtlCode returns the value of the *Function* part of the given IOCTL_*XXX* code.

Comments

See *Defining I/O Control Codes* in Chapter 13 for more information about the layout of IOCTL_*XXX* codes and using the CTL_CODE macro.

Callers of **IoGetFunctionCodeFromCtlCode** must be running at IRQL <= DISPATCH_LEVEL.

See Also

CTL_CODE, **IoBuildDeviceIoControlRequest**

IoGetInitialStack

```
PVOID
  IoGetInitialStack( );
```

IoGetInitialStack returns the base address of the current thread's stack.

Include

ntddk.h

Return Value

IoGetInitialStack returns the initial base address of the current thread's stack.

Comments

Highest-level drivers can call this routine, particularly file systems attempting to determine whether they've been passed a pointer to a location on the current thread's stack.

Callers of **IoGetInitialStack** must be running at IRQL < DISPATCH_LEVEL.

See Also

IoGetRemainingStackSize, **IoGetStackLimits**

IoGetNextIrpStackLocation

```
PIO_STACK_LOCATION
  IoGetNextIrpStackLocation(
  IN PIRP  Irp
  );
```

IoGetNextIrpStackLocation gives a higher level driver access to the next-lower driver's I/O stack location in an IRP so the caller can set it up for the lower driver.

Parameters

Irp

Points to the IRP.

Include

wdm.h or *ntddk.h*

Return Value

IoGetNextIrpStackLocation returns a pointer to the next-lower-level driver's I/O stack location in the given IRP.

Comments

Each driver that passes IRPs on to lower drivers must set up the stack location for the next lower driver. A driver calls **IoGetNextIrpStackLocation** to get a pointer to the next-lower driver's I/O stack location.

If a driver is passing the same parameters that it received to the next-lower driver, such a driver can call **IoCopyCurrentIrpStackLocationToNext** or **IoSkipCurrentIrpStackLocation** instead of getting a pointer to the next-lower stack location and copying the parameters manually.

Callers of **IoGetNextIrpStackLocation** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IO_STACK_LOCATION, **IoCallDriver**, **IoGetCurrentIrpStackLocation**, **IoCopyCurrentIrpStackLocationToNext**, **IoSetNextIrpStackLocation**, **IoSkipCurrentIrpStackLocation**

IoGetRelatedDeviceObject

```
PDEVICE_OBJECT
  IoGetRelatedDeviceObject(
  IN PFILE_OBJECT  FileObject
  );
```

Given a file object, **IoGetRelatedDeviceObject** returns a pointer to the corresponding device object.

Parameters

FileObject

Points to the file object.

Include

wdm.h or *ntddk.h*

Return Value

IoGetRelatedDeviceObject returns a pointer to the device object.

Comments

When called on a file object that represents the underlying storage device, **IoGetRelatedDeviceObject** returns the highest-level device object in the storage device stack. To obtain the highest-level device object in the file system driver stack, drivers must call **IoGetRelatedDeviceObject** on a file object that represents the file system's driver stack, and the file system must currently be mounted. (Otherwise, the storage device stack is traversed instead of the file system stack.)

To ensure that the file system is mounted on the storage device, the driver must have specified an appropriate access mask, such as FILE_READ_DATA or FILE_WRITE_ATTRIBUTES, when opening the file or device represented by the file object. Specifying FILE_READ_ATTRIBUTES does not cause the file system to be mounted.

The caller must be running at IRQL <= DISPATCH_LEVEL. Usually, callers of this routine are running at IRQL PASSIVE_LEVEL.

See Also

IoGetDeviceObjectPointer

IoGetRemainingStackSize

```
ULONG
  IoGetRemainingStackSize( );
```

IoGetRemainingStackSize returns the current amount of available kernel-mode stack space.

Include

ntddk.h

Return Value

IoGetRemainingStackSize returns the number of bytes of stack space in the current thread context.

Comments

Highest-level drivers, such as file systems, can call this routine, particularly drivers that use recursive code paths. Such a driver would call **IoGetRemainingStackSize** before launching a recursion to determine whether it should continue processing on an alternate code path.

Callers of **IoGetRemainingStackSize** must be running at IRQL < DISPATCH_LEVEL.

See Also

IoGetInitialStack, **IoGetStackLimits**

IoGetStackLimits

```
VOID
  IoGetStackLimits(
  OUT PULONG  LowLimit,
  OUT PULONG  HighLimit
  );
```

IoGetStackLimits returns the boundaries of the current thread's stack frame.

Parameters

LowLimit

Points to a caller-supplied variable in which this routine returns the lower offset of the current thread's stack frame.

HighLimit

Points to a caller-supplied variable in which this routine returns the higher offset of the current thread's stack frame.

Include

ntddk.h

Comments

Highest-level drivers can call this routine, particularly file systems that have been passed a pointer to a location on the current thread's stack.

Callers of **IoGetStackLimits** must be running at IRQL < DISPATCH_LEVEL.

See Also

IoGetInitialStack, IoGetRemainingStackSize

IoInitializeDpcRequest

```
VOID
  IoInitializeDpcRequest(
  IN PDEVICE_OBJECT  DeviceObject,
  IN PIO_DPC_ROUTINE  DpcRoutine
  );
```

IoInitializeDpcRequest registers a driver-supplied DpcForIsr routine when a device driver initializes.

Parameters

DeviceObject

Points to the device object representing the physical device that generates interrupts.

DpcRoutine

Points to the driver-supplied DpcForIsr routine, which is declared as follows:

```
VOID
(*PIO_DPC_ROUTINE)(
    IN PKDPC Dpc,
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    );
```

Include

wdm.h or *ntddk.h*

Comments

IoInitializeDpcRequest associates a driver-supplied DpcForIsr routine with a given device object so the driver's ISR can call **IoRequestDpc** to queue the DpcForIsr. This routine completes interrupt-driven I/O operations at a lower IRQL than that of the ISR. For details, see *DpcForIsr Routine and CustomDpc Routines* in the *Kernel-Mode Drivers Design Guide*.

PnP drivers call **IoInitializeDpcRequest** from the AddDevice routine.

Callers of **IoInitializeDpcRequest** must be running at IRQL PASSIVE_LEVEL.

It is possible to call **KeInitializeDpc** to initialize another DPC at IRQL <= DISPATCH_LEVEL.

See Also

IoRequestDpc, **KeInitializeDpc**

IoInitializeIrp

```
VOID
  IoInitializeIrp(
  IN OUT PIRP  Irp,
  IN USHORT  PacketSize,
  IN CCHAR  StackSize
  );
```

IoInitializeIrp initializes a given IRP that was allocated by the caller.

Parameters

Irp

Points to the IRP to be initialized.

PacketSize

Specifies the size in bytes of the IRP.

StackSize

Specifies the number of stack locations in the IRP.

Include

wdm.h or *ntddk.h*

Comments

Drivers use **IoInitializeIrp** to initialize IRPs the driver allocated as raw memory. Drivers must not call **IoInitializeIrp** on an IRP that was allocated by **IoAllocateIrp**. Drivers must not use **IoInitializeIrp** to reinitialize an already initialized IRP. Instead, use **IoReuseIrp**.

If the driver associates an MDL with the IRP it allocated, the driver is responsible for releasing the MDL when the IRP is completed.

An intermediate or highest-level driver also can call **IoBuildDeviceIoControlRequest**, **IoBuildAsynchronousFsdRequest**, or **IoBuildSynchronousFsdRequest** to set up requests it sends to lower-level drivers. Only a highest-level driver can call **IoMakeAssociatedIrp**.

Callers of **IoInitializeIrp** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateIrp, IoAllocateMdl, IoBuildPartialMdl, IoFreeIrp, IoFreeMdl, IoReuseIrp, IoSetNextIrpStackLocation, IoSizeOfIrp, IRP

IoInitializeRemoveLock

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoInitializeRemoveLock*** in that book for a full reference.

IoInitializeRemoveLockEx

This routine is documented in the Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoInitializeRemoveLockEx*** in that book for a full reference.

IoInitializeTimer

```
NTSTATUS
  IoInitializeTimer(
  IN PDEVICE_OBJECT  DeviceObject,
  IN PIO_TIMER_ROUTINE  TimerRoutine,
  IN PVOID  Context
  );
```

IoInitializeTimer sets up a driver-supplied IoTimer routine associated with a given device object.

Parameters

DeviceObject

Points to a device object representing a device on which I/O operations can time out.

TimerRoutine

Points to the driver-supplied IoTimer routine, which is declared as follows:

```
VOID
(*PIO_TIMER_ROUTINE) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PVOID Context
    );
```

Context

Points to the driver-determined context with which its IoTimer routine will be called.

Include

wdm.h or *ntddk.h*

Return Value

IoInitializeTimer returns STATUS_SUCCESS if the IoTimer routine is set up.

Comments

A driver's IoTimer routine is called once per second after the driver enables the timer by calling **IoStartTimer**.

The driver can disable the timer by calling **IoStopTimer** and can re-enable it again with **IoStartTimer**.

The driver's IoTimer routine is called at IRQL DISPATCH_LEVEL and therefore must not contain pageable code.

When the timer is running, the I/O Manager calls the driver-supplied IoTimer routine once per second. Drivers whose time-out routines should be called at variable intervals or at intervals of finer granularity can set up a CustomTimerDpc routine and use the **Ke..Timer** routines.

Callers of **IoInitializeTimer** must be running at IRQL PASSIVE_LEVEL.

See Also

IoStartTimer, **IoStopTimer**, **KeInitializeTimer**, **KeSetTimer**

IoInvalidateDeviceRelations

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoInvalidateDeviceRelations*** in that book for a full reference.

IoInvalidateDeviceState

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoInvalidateDeviceState*** in that book for a full reference.

IoIsErrorUserInduced

```
BOOLEAN
  IoIsErrorUserInduced(
  IN NTSTATUS  Status
  );
```

IoIsErrorUserInduced determines whether an I/O error encountered while processing a request to a removable-media device was caused by the user.

Parameters

Status

Specifies the current NTSTATUS value, usually within the driver's DpcForIsr routine.

Include

wdm.h or *ntddk.h*

Return Value

IoIsErrorUserInduced returns TRUE if an I/O request failed because of a user-induced error.

Comments

This routine indicates whether an I/O request failed for one of the following user-correctable conditions:

STATUS_DEVICE_NOT_READY
STATUS_IO_TIMEOUT
STATUS_MEDIA_WRITE_PROTECTED
STATUS_NO_MEDIA_IN_DEVICE
STATUS_UNRECOGNIZED_MEDIA
STATUS_VERIFY_REQUIRED
STATUS_WRONG_VOLUME

If **IoIsErrorUserInduced** returns TRUE, the removable-media driver must call **IoSetHardErrorOrVerifyDevice** before completing the IRP.

Callers of **IoIsErrorUserInduced** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoSetHardErrorOrVerifyDevice, **IoAllocateErrorLogEntry**, **IoWriteErrorLogEntry**

IoIsWdmVersionAvailable

```
BOOLEAN
  IoIsWdmVersionAvailable(
  IN UCHAR  MajorVersion,
  IN UCHAR  MinorVersion
  );
```

IoIsWdmVersionAvailable checks whether a given WDM version is supported by the operating system.

Parameters

MajorVersion

Specifies the major version number of WDM that is requested.

MinorVersion

Specifies the minor version number of WDM that is requested.

Include

wdm.h or *ntddk.h*

Return Value

IoIsWdmVersionAvailable returns TRUE if the version of WDM that the operating system provides is greater than or equal to the version number of WDM being requested.

Comments

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

IoMakeAssociatedIrp

```
PIRP
  IoMakeAssociatedIrp(
  IN PIRP  Irp,
  IN CCHAR  StackSize
  );
```

IoMakeAssociatedIrp allocates and initalizes an IRP to be associated with a master IRP sent to a highest-level driver, allowing the caller to split the original request and send associated IRPs on to lower-level drivers.

Parameters

Irp

Points to the master IRP that was input to a highest-level driver's Dispatch routine.

StackSize

Specifies the number of stack locations to be allocated for the associated IRP. The value must be at least equal to the **StackSize** of the next-lower driver's device object, but the associated IRP can have an additional stack location for the caller.

Include

ntddk.h

Return Value

IoMakeAssociatedIrp returns a pointer to the associated IRP or returns a NULL pointer if an IRP cannot be allocated.

Comments

Only a highest-level driver can call this routine.

The I/O Manager completes the master IRP automatically when lower drivers have completed all associated IRPs as long as the caller has not set its IoCompletion routine in an associated IRP and returned STATUS_MORE_PROCESSING_REQUIRED from its IoCompletion routine. In these circumstances, the caller must explicitly complete the master IRP when that driver has determined that all associated IRPs were completed.

Only the master IRP is associated with a thread; associated IRPs are not. For this reason, the I/O Manager cannot call Cancel routines for associated IRPs when a thread exits. When the master IRP's thread exits, the I/O Manager calls the master IRP's cancel routine. The Cancel routine is responsible for tracking down all associated IRPs and calling **IoCancelIrp** to cancel them.

Callers of **IoMakeAssociatedIrp** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateIrp, **IoBuildAsynchronousFsdRequest**, **IoBuildDeviceIoControlRequest**, **IoBuildSynchronousFsdRequest**, **IoCallDriver**, **IoSetCompletionRoutine**, IRP

IoMapTransfer

```
PHYSICAL_ADDRESS
  IoMapTransfer(
  IN PADAPTER_OBJECT  AdapterObject,
  IN PMDL  Mdl,
  IN PVOID  MapRegisterBase,
  IN PVOID  CurrentVa,
  IN OUT PULONG  Length,
  IN BOOLEAN  WriteToDevice
  );
```

IoMapTransfer is obsolete and exported only to support existing drivers. See **MapTransfer** instead.

IoMapTransfer sets up a number of map registers (up to the number returned by HalGetAdapter) for the given adapter object to map a transfer from a locked-down buffer specified by *Mdl*, the given *CurrentVa* into the MDL, *Length* in bytes to be transferred, and transfer direction.

Parameters

AdapterObject

Points to the adapter object pointer returned by **HalGetAdapter** and already passed in a call to **IoAllocateAdapterChannel** for the current IRP's transfer request.

Mdl

Points to the MDL describing the buffer either in the current IRP at **MdlAddress** or the MDL that the driver of a slave device using auto-initialize mode set up to describe the driver's common buffer.

MapRegisterBase

Points to the handle returned by **IoAllocateAdapterChannel**, which the driver already called for the current IRP.

CurrentVa

Points to the current virtual address in the buffer, described by the *Mdl*, to be mapped for a DMA transfer operation.

Length

Specifies the length, in bytes, to be mapped. If the driver indicated that its device was a busmaster with scatter/gather support when it called **HalGetAdapter**, the value of *Length* on return from **IoMapTransfer** indicates how many bytes were mapped. Otherwise, the input and output values of *Length* are identical.

WriteToDevice

Indicates the direction of the transfer operation: TRUE for a transfer from the locked-down buffer to the device.

Include

wdm.h or *ntddk.h*

Return Value

IoMapTransfer returns the logical address of the region mapped, which the driver of a bus-master adapter can use. Drivers of devices that use a system DMA controller cannot use this value and should ignore it.

Comments

The *AdapterObject* must have already been allocated to the driver in a preceding call to **IoAllocateAdapterChannel**.

The number of map registers that can be set up cannot exceed the maximum returned when the driver called **HalGetAdapter**.

The initial *CurrentVa* for the start of a packet-based DMA transfer can be obtained by calling **MmGetMdlVirtualAddress**. However, the value returned is an index into the *Mdl*, rather than a valid virtual address. If the driver must split a large transfer request into more than one DMA operation, *CurrentVa* and *Length* must be updated for each DMA operation.

The driver of a busmaster device with scatter/gather support can use the returned logical address and updated *Length* value to build a scatter/gather list, calling **IoMapTransfer** repeatedly until it has used all available map registers for the transfer operation.

Callers of **IoMapTransfer** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ADDRESS_AND_SIZE_TO_SPAN_PAGES, **HalAllocateCommonBuffer**, **HalGetAdapter**, **IoAllocateAdapterChannel**, **IoFlushAdapterBuffers**, **IoFreeAdapterChannel**, **IoFreeMapRegisters**, **KeFlushIoBuffers**, **MmGetMdlVirtualAddress**

IoMarkIrpPending

```
VOID
  IoMarkIrpPending(
  IN OUT PIRP  Irp
  );
```

IoMarkIrpPending marks the given IRP, indicating that a driver's Dispatch routine returned STATUS_PENDING because further processing is required by other driver routines.

Parameters

Irp

Points to the IRP to be marked as pending.

Include

wdm.h or *ntddk.h*

Comments

Unless a driver calls **IoCompleteRequest** from its Dispatch routine with a given IRP or passes the IRP on to lower drivers, it must call **IoMarkIrpPending** with the IRP. Otherwise, the I/O Manager attempts to complete the IRP as soon as the Dispatch routine returns control.

If a driver queues incoming IRPs, it should call **IoMarkIrpPending** before it queues each IRP. Otherwise, an IRP could be dequeued, completed by another driver routine, and freed by the system before the call to **IoMarkIrpPending** occurs, thereby causing a crash.

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.

A routine that calls **IoMarkIrpPending** must return STATUS_PENDING.

Callers of **IoMarkIrpPending** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoCompleteRequest, **IoStartPacket**, IRP

IoOpenDeviceInterfaceRegistryKey

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoOpenDeviceInterfaceRegistryKey*** in that book for a full reference.

IoOpenDeviceRegistryKey

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoOpenDeviceRegistryKey*** in that book for a full reference.

IoQueryDeviceDescription

```
NTSTATUS
  IoQueryDeviceDescription(
  IN PINTERFACE_TYPE  BusType  OPTIONAL,
  IN PULONG  BusNumber  OPTIONAL,
  IN PCONFIGURATION_TYPE  ControllerType  OPTIONAL,
  IN PULONG  ControllerNumber  OPTIONAL,
  IN PCONFIGURATION_TYPE  PeripheralType  OPTIONAL,
  IN PULONG  PeripheralNumber  OPTIONAL,
  IN PIO_QUERY_DEVICE_ROUTINE  CalloutRoutine,
  IN PVOID  Context
  );
```

IoQueryDeviceDescription 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.

This routine is obsolete and is exported only to support existing drivers.

Drivers that require hardware configuration information should use **IoGetDeviceProperty** instead.

Parameters

BusType

Specifies the type of bus searched for in the registry's hardware description tree which can be one of the following values: **Internal**, **Isa**, **Eisa**, **MicroChannel TurboChannel,** or **PCIBus**. However, additional types of buses will be supported in future versions of the operating system. The upper bound on the types supported is always **Maximum InterfaceType**.

BusNumber

Specifies the zero-based and system-assigned number of the bus. This parameter is optional. If *BusType* is supplied but no specific *BusNumber* is specified, information on all buses of type *BusType* is returned.

ControllerType

Specifies the type of controller for which to return information. It can be one of the following: **DiskController**, **TapeController**, **CdRomController**, **WormController**, **SerialController**, **NetworkController**, **DisplayController**, **ParallelController**, **PointerController**, **KeyboardController**, **AudioController**, or **OtherController**. If no *ControllerType* or *PeripheralType* value is specified, only bus information is returned.

ControllerNumber

Specifies the zero-based number of the controller. This parameter is optional. If *ControllerType* is supplied but a specific *ControllerNumber* is not, information on all controllers of type *ControllerType* is returned.

PeripheralType

Specifies the type of peripheral for which to return information. It can be one of the following: **DiskPeripheral**, **FloppyDiskPeripheral**, **TapePeripheral**, **ModemPeripheral**, **MonitorPeripheral**, **PrinterPeripheral**, **PointerPeripheral**, **KeyboardPeripheral**, **TerminalPeripheral**, **OtherPeripheral**, **LinePeripheral**, or **NetworkPeripheral**. If no peripheral type is specified, only bus information and controller information are returned.

PeripheralNumber

Specifies the zero-based number of the peripheral. This parameter is optional. If *PeripheralType* is supplied but a specific *PeripheralNumber* is not, information on all peripherals of type *PeripheralType* is returned.

CalloutRoutine

Points to a driver-supplied routine to be called when the requested information has been located. This routine is declared as follows:

```
NTSTATUS
(*PIO_QUERY_DEVICE_ROUTINE) (
    IN PVOID Context,
    IN PUNICODE_STRING PathName,
    IN INTERFACE_TYPE BusType,
    IN ULONG BusNumber,
    IN PKEY_VALUE_FULL_INFORMATION *BusInformation,
    IN CONFIGURATION_TYPE ControllerType,
    IN ULONG ControllerNumber,
    IN PKEY_VALUE_FULL_INFORMATION *ControllerInformation,
    IN CONFIGURATION_TYPE PeripheralType,
    IN ULONG PeripheralNumber,
    IN PKEY_VALUE_FULL_INFORMATION *PeripheralInformation
    );
```

Context

Points to a context area that is passed to *CalloutRoutine*.

Include

ntddk.h

Return Value

IoQueryDeviceDescription returns the STATUS_*XXX* returned by the callout routine.

Comments

This routine queries the registry for description(s) of the given bus type and number, controller type and number, and/or peripheral type and number. The information retrieved is passed to a driver-supplied ConfigCallback routine.

While the bus, controller, and peripheral parameters are each optional, the caller must supply at least one type parameter.

On entry, the driver's ConfigCallback routine is given pointers to registry keys for bus, controller, and/or peripheral information. Each such pointer is actually a pointer to an array of IO_QUERY_DEVICE_DATA_FORMAT pointers identified as follows:

IoQueryDeviceIdentifier
IoQueryDeviceConfigurationData
IoQueryDeviceComponentInformation

When the ConfigCallback routine returns control, these pointers become invalid. The driver's ConfigCallback routine should save pertinent information about the I/O ports or device memory, the bus-relative interrupt vector or IRQL, and/or the DMA channel or port, that is available in the registry for the DriverEntry routine to use in subsequent calls to **HalTranslateBusAddress** (and possibly **MmMapIoSpace**), **HalGetInterruptVector**, and/or **HalGetAdapter**.

As an alternative, a driver can call **HalGetBusData** or **HalGetBusDataByOffset** to locate its device(s) and to retrieve bus-relative configuration information. Then, the driver can call **IoAssignResources**, which checks the input resource list against the hardware configuration information in the registry and also encapsulates most of the functionality of **IoReportResourceUsage**. As an alternative, drivers of PCI-type devices can call **HalAssignSlotResources**.

Callers of **IoQueryDeviceDescription** must be running at IRQL PASSIVE_LEVEL.

See Also

IoGetDeviceProperty, **IoGetDmaAdapter**, **MmMapIoSpace**

IoQueueWorkItem

```
VOID
  IoQueueWorkItem(
  IN PIO_WORKITEM pIOWorkItem,
  IN PIO_WORKITEM_ROUTINE Routine,
  IN WORK_QUEUE_TYPE QueueType,
  IN PVOID Context
  );
```

IoQueueWorkItem inserts the specified work item into a queue from which a system worker thread removes the item and gives control to the specified callback routine.

Parameters

pIOWorkItem

Points to the IO_WORKITEM structure returned by a previous call to **IoAllocateWorkItem**.

Routine

Points to the routine that will be called to process the work item. This routine will be called in the context of a system thread. This routine is declared as follows:

```
VOID
(*PIO_WORKITEM_ROUTINE) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PVOID Context
    );
```

QueueType

Specifies the type of work queue that the work item should be inserted. *QueueType* can be either of the following:

CriticalWorkQueue

Insert the work item into the queue from which a system thread with a real-time priority attribute will process the work item.

DelayedWorkQueue

Insert the work item into the queue from which a system thread with a variable priority attribute will process the work item.

The *QueueType* value **HyperCriticalWorkQueue** is reserved for system use.

Context

Points to a caller-supplied context area to be passed through to the callback routine.

Include

ntddk.h or *wdm.h*

Comment

Highest-level drivers can call **IoQueueWorkItem**.

The callback is run within a system thread context at IRQL PASSIVE_LEVEL. This caller-supplied routine is responsible for calling **IoFreeWorkItem** to reclaim the storage allocated for the work item.

A driver must not wait for its callback routine to complete an operation if it is already holding one synchronization object and might attempt to acquire another. For example, a driver should release any currently held semaphores, mutexes, resource variables, and so forth *before* it calls **IoQueueWorkItem**. Releasing synchronization resources before queuing a synchronous worker-thread operation prevents deadlocks.

The value of *QueueType* determines the runtime priority at which the callback routine is run, as follows:

- If the callback runs in the system thread with a real-time priority attribute, the callback routine cannot be preempted except by threads with higher real-time priorities.
- If the callback runs in the system thread with a variable priority attribute, the callback can be preempted by threads with higher variable and real-time priorities, and the callback is scheduled to run round-robin with other threads of the same priority for a quantum each.

Threads at either priority remain interruptible.

Callers of **IoQueueWorkItem** must be running at IRQL <= DISPATCH_LEVEL.

IoQueueWorkItem should be used instead of **ExQueueWorkItem** because **IoQueueWorkItem** will ensure that the device object associated with the specified work item is available for the processing of the work item.

See Also

IoAllocateWorkItem, IoFreeWorkItem

IoRaiseHardError

```
VOID
  IoRaiseHardError(
  IN PIRP  Irp,
  IN PVPB  Vpb  OPTIONAL,
  IN PDEVICE_OBJECT  RealDeviceObject
  );
```

IoRaiseHardError causes a popup to be displayed that warns the user that a device I/O error has occurred. The I/O error might indicate that a physical device is failing.

Parameters

Irp

Points to the IRP that failed because of a device I/O error.

Vpb

Points to the volume parameter block (VPB), if any, for the mounted file object. This parameter is NULL if no VPB is associated with the device object.

RealDeviceObject

Points to the device object that represents the physical device on which the I/O operation failed.

Include

ntddk.h

Comments

Highest-level drivers, particularly file system drivers, call **IoRaiseHardError**.

*Warning: Because **IoRaiseHardError** uses a normal kernel APC to create a user popup, a deadlock can occur if normal kernel APCs are disabled when a device error occurs.* For example:

- An upper-level filter driver calls **KeEnterCriticalRegion** (which disables normal kernel APCs) and sends an I/O request to a a file system driver. The filter driver waits on the completion of the request by the file system driver before the filter driver calls **KeLeaveCriticalRegion** (which re-enables normal kernel APCs).
- An error occurs on the file system and the file system driver calls **IoRaiseHardError** to report the error to the user. The file system driver waits on the popup.
- Deadlock now exists: The normal kernel APC created by **IoRaiseHardError** to create the popup waits for normal kernel APCs to be enabled. The file system waits on the popup before it completes the I/O request. The filter driver waits on completion of the I/O request before it calls **KeLeaveCriticalRegion** (which re-enables normal kernel APCs).

The behavior of this routine is dependent of the current state of hard errors for the running thread. If hard errors have been disabled by calling **IoSetThreadHardErrorMode**, this routine completes the IRP specified by *Irp* without transferring any data into user buffers. In addition, no message is sent to notify the user of this failure.

Callers of **IoRaiseHardError** must be running at IRQL < DISPATCH_LEVEL.

See Also

IoGetRelatedDeviceObject, IoSetHardErrorOrVerifyDevice, IoSetThreadHardErrorMode

IoRaiseInformationalHardError

```
BOOLEAN
  IoRaiseInformationalHardError(
  IN NTSTATUS  ErrorStatus,
  IN PUNICODE_STRING  String  OPTIONAL,
  IN PKTHREAD  Thread  OPTIONAL
  );
```

IoRaiseInformationalHardError sends a popup to the user, warning about a device I/O error that indicates why a user I/O request failed.

Parameters

ErrorStatus

Identifies the error status (IO_ERR_*XXX*).

String

Points to a Unicode string, which provides additional information about the error. Some NT status codes require a string parameter, such as a file or directory name. If the *ErrorStatus* does not require a parameter, then *String* is NULL.

Thread

Points to the thread whose IRP was failed due to the error.

Include

ntddk.h

Return Value

IoRaiseInformationalHardError returns TRUE if the popup was successfully queued. This routine returns FALSE if popups are disabled for *Thread*, a pool allocation failed, too many popups are already queued, or an equivalent popup is already pending a user response (such as waiting for the user to press RETURN).

Comments

IoRaiseInformationalHardError takes a system-defined NT error value as a parameter. Driver writers can use the event log APIs to communicate driver-defined event strings to the user.

If hard errors have been disabled for this thread, by calling **IoSetThreadHardErrorMode**, this routine returns FALSE.

Callers of **IoRaiseInformationalHardError** must be running at IRQL < DISPATCH_LEVEL.

See Also

IoSetHardErrorOrVerifyDevice, **IoSetThreadHardErrorMode**, **PsGetCurrentThread**

IoReadPartitionTable

```
NTSTATUS
  IoReadPartitionTable(
  IN PDEVICE_OBJECT  DeviceObject,
  IN ULONG  SectorSize,
  IN BOOLEAN  ReturnRecognizedPartitions,
  OUT struct _DRIVE_LAYOUT_INFORMATION  **PartitionBuffer
  );
```

IoReadPartitionTable reads a list of partitions on a disk having a specified sector size and creates an entry in the partition list for each recognized partition.

Parameters

DeviceObject

Points to the device object for the disk whose partitions are to be read.

SectorSize

Specifies the size of the sectors on the disk.

ReturnRecognizedPartitions

Indicates whether only recognized partitions or all partition entries should be returned.

PartitionBuffer

Is a pointer to an uninitialized address. If successful, **IoReadPartitionTable** allocates the memory for this buffer from nonpaged pool and returns the drive layout information in it.

Include

ntddk.h

Return Value

This routine returns a value of STATUS_SUCCESS if at least one sector table was read. Otherwise, it returns an error status and sets the pointer at *PartitionBuffer* to NULL.

Comments

Disk device drivers call this routine during driver initialization.

It is the responsibility of the caller to deallocate the *PartitionBuffer* that was allocated by this routine with **ExFreePool**.

The algorithm used by this routine is determined by the Boolean value *ReturnRecognized-Partitions*:

- Read each partition table and, for each valid and recognized partition found, fill in a partition information entry. Extended partitions are located in order to find other partition tables, but no entries are built for them.
- Read each partition table and, for each and every entry, fill in a partition information entry. Extended partitions are located to find each partition on the disk, and entries are built for these as well.

The drive layout structure contains a variable-sized array of partition information elements, defined as follows:

```
typedef struct _DRIVE_LAYOUT_INFORMATION {
    ULONG PartitionCount;
    ULONG Signature;                // of disk
    PARTITION_INFORMATION PartitionEntry[1];
} DRIVE_LAYOUT_INFORMATION, *PDRIVE_LAYOUT_INFORMATION;

typedef strtuct _PARTITION_INFORMATION {
    LARGE_INTEGER StartingOffset;
    LARGE_INTEGER PartitionLength;
    ULONG HiddenSectors;
    ULONG PartitionNumber;
    UCHAR PartitionType;            // 12-bit FAT etc.
    BOOLEAN BootIndicator;
    BOOLEAN RecognizedPartition;
    BOOLEAN RewritePartition;
} PARTITION_INFORMATION, *PPARTITION_INFORMATION;
```

For the currently defined *PartitionType* values, see the Win32 SDK.

Note that disk drivers also use the DRIVE_LAYOUT_INFORMATION structure to return and set partition information in response to IRP_MJ_DEVICE_CONTROL requests with the following I/O control codes:

IOCTL_DISK_GET_PARTITION_INFO
IOCTL_DISK_GET_DRIVE_LAYOUT
IOCTL_DISK_SET_DRIVE_LAYOUT

Callers of **IoReadPartitionTable** must be running at IRQL PASSIVE_LEVEL.

See Also

IOCTL_DISK_GET_PARTITION_INFO, IOCTL_DISK_GET_DRIVE_LAYOUT, IOCTL_DISK_SET_DRIVE_LAYOUT, **IoSetPartitionInformation**, **IoWrite-PartitionTable**

IoRegisterDeviceInterface

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoRegisterDeviceInterface*** in that book for a full reference.

IoRegisterDriverReinitialization

```
VOID
  IoRegisterDriverReinitialization(
  IN PDRIVER_OBJECT  DriverObject,
  IN PDRIVER_REINITIALIZE  DriverReinitializationRoutine,
  IN PVOID  Context
  );
```

IoRegisterDriverReinitialization is called by a driver during its initialization or reinitialization to register its Reinitialize routine to be called again before the driver's and, possibly the system's, initialization is complete.

Parameters

DriverObject

Points to the driver object that was input to the **DriverEntry** routine.

DriverReinitializationRoutine

Specifies the entry point for the driver-supplied Reinitialize routine, which is declared as follows:

```
VOID
(*PDRIVER_REINITIALIZE)(
    IN PDRIVER_OBJECT DriverObject,
    IN PVOID Context,
    IN ULONG Count
    );
```

Context

Points to the context to be passed to the driver's Reinitialize routine.

Include

ntddk.h

Comments

A driver can call this routine only if its **DriverEntry** routine will return STATUS_SUCCESS. If the driver-supplied Reinitialize routine must use the registry, the **DriverEntry** routine should include a copy of the string to which *RegistryPath* points as part of the context passed to the Reinitialize routine in this call.

If the driver is loaded dynamically, it is possible for this to occur during a normally running system, so all references to the reinitialization queue must be synchronized.

The *Count* input to a *DriverReinitializationRoutine* indicates how many times this routine has been called, including the current call.

The **DriverEntry** routine can call **IoRegisterDriverReinitialization** only once. If the Reinitialize routine should be run again after any other drivers' Reinitialize routines have returned control, the Reinitialize routine also can call **IoRegisterDriverReinitialization** as many times as the driver's Reinitialize routine should be run.

Usually, a driver with a Reinitialize routine is a higher-level driver that controls both PnP and legacy devices. Such a driver must not only create 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 detect. A driver can use a Reinitialize routine to create those device objects and layer the driver over the next-lower driver for the underlying device.

Callers of **IoRegisterDriverReinitialization** must be running at IRQL PASSIVE_LEVEL.

See Also

DRIVER_OBJECT

IoRegisterPlugPlayNotification

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoRegisterPlugPlayNotification*** in that book for a full reference.

IoRegisterShutdownNotification

```
NTSTATUS
  IoRegisterShutdownNotification(
  IN PDEVICE_OBJECT  DeviceObject
  );
```

IoRegisterShutdownNotification registers a driver-supplied DispatchShutdown routine to be called before system shutdown. The driver gains control before the system is fully shut down.

Parameters

DeviceObject

Points to a device object.

Include

ntddk.h

Return Value

IoRegisterShutdownNotification returns STATUS_SUCCESS when the shutdown routine is registered.

Comments

Only highest-level drivers chained above an underlying nonmass-storage device can register a shutdown routine by calling **IoRegisterShutdownNotification**. For mass-storage drivers, an FSD handles shutdown requests and calls lower drivers to flush cached or buffered data out to the device before the system is shut down.

IoRegisterShutdownNotification places the given *DeviceObject* on the shutdown notification queue, so that its DispatchShutdown routine can be called before the system shuts down. A driver can call **IoRegisterShutdownNotification** multiple times.

The registered DispatchShutdown routine is called before the Power Manager sends an IRP_MN_SET_POWER request for **PowerSystemS5**. The DispatchShutdown routine is not called for transitions to any other power states.

A driver can make no assumptions about the order in which its DispatchShutdown routine will be called in relation to other such routines or to other shutdown activities.

A PnP driver might register a shutdown routine so that it can do certain tasks before system shutdown starts, such as locking down code.

Callers of **IoRegisterShutdownNotification** must be running at IRQL PASSIVE_LEVEL.

See Also

IoUnregisterShutdownNotification

IoReleaseCancelSpinLock

```
VOID
  IoReleaseCancelSpinLock(
  IN KIRQL  Irql
  );
```

IoReleaseCancelSpinLock releases the cancel spin lock after the driver has changed the cancelable state of an IRP. This routine also releases the cancel spin lock from the driver's Cancel routine.

Parameters

Irql

Points to the IRQL returned by **IoAcquireCancelSpinLock**.

Include

wdm.h or *ntddk.h*

Comments

This routine is a reciprocal to **IoAcquireCancelSpinLock**.

The holder of the cancel spin lock executes at DISPATCH_LEVEL IRQL after calling **IoAcquireCancelSpinLock**. **IoReleaseCancelSpinLock** restores the original IRQL of its caller.

Callers of **IoReleaseCancelSpinLock** must be running at IRQL DISPATCH_LEVEL.

See Also

IoAcquireCancelSpinLock, IoSetCancelRoutine

IoReleaseRemoveLock

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoReleaseRemoveLock*** in that book for a full reference.

IoReleaseRemoveLockEx

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoReleaseRemoveLockEx*** in that book for a full reference.

IoReleaseRemoveLockAndWait

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoReleaseRemoveLockAndWait*** in that book for a full reference.

IoReleaseRemoveLockAndWaitEx

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoReleaseRemoveLockAndWaitEx*** in that book for a full reference.

IoRemoveShareAccess

```
VOID
  IoRemoveShareAccess(
  IN PFILE_OBJECT  FileObject,
  IN OUT PSHARE_ACCESS  ShareAccess
  );
```

IoRemoveShareAccess removes the access and share-access information for a given open instance of a file object.

Parameters

FileObject

Points to the file object, which usually is being closed by the current thread.

ShareAccess

Points to the share-access structure that describes how the open file object is currently being accessed.

Include

ntddk.h

Comments

This routine is a reciprocal to **IoUpdateShareAccess**.

IoRemoveShareAccess is not an atomic operation. Therefore, drivers calling this routine must protect the shared file object passed to **IoRemoveShareAccess** by means of some kind of lock, such as a mutex or a resource lock, in order to prevent corruption of the shared access counts.

Callers of **IoRemoveShareAccess** must be running at IRQL PASSIVE_LEVEL and in the context of the thread that requested that the *FileObject* be closed.

See Also

IoCheckShareAccess, **IoSetShareAccess**, **IoUpdateShareAccess**

IoReportDetectedDevice

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoReportDetectedDevice*** in that book for a full reference.

IoReportResourceForDetection

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoReportResourceForDetection*** in that book for a full reference.

IoReportResourceUsage

```
NTSTATUS
  IoReportResourceUsage(
  IN PUNICODE_STRING  DriverClassName  OPTIONAL,
  IN PDRIVER_OBJECT  DriverObject,
  IN PCM_RESOURCE_LIST  DriverList  OPTIONAL,
  IN ULONG  DriverListSize  OPTIONAL,
  IN PDEVICE_OBJECT  DeviceObject,
  IN PCM_RESOURCE_LIST  DeviceList  OPTIONAL,
  IN ULONG  DeviceListSize  OPTIONAL,
  IN BOOLEAN  OverrideConflict,
  OUT PBOOLEAN  ConflictDetected
  );
```

IoReportResourceUsage 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.

This routine is *obsolete*, and is supported only for existing drivers. 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.

Parameters

DriverClassName

Points to a buffered Unicode String that describes the class of driver under which the resource information should be stored. A default type **Other** is used if none is given, and a new key is created in the registry if a unique name is supplied.

DriverObject

Points to the driver object that was input to the DriverEntry routine.

DriverList

Points to the driver's resource list if the driver claims the same resources for all its devices. This pointer is NULL if the driver claims resources for its devices separately.

DriverListSize

Specifies the size in bytes of the driver's resource list if the *DriverList* pointer is nonNULL; otherwise, zero.

DeviceObject

Points to the driver-created device object representing a device for which the driver is attempting to claim resources.

DeviceList

Points to the device's resource list, if the driver claims resources separately for each of its devices.

DeviceListSize

Specifies the size in bytes of the *DeviceList* if the *DeviceList* pointer is nonNULL; otherwise zero.

OverrideConflict

Specifies a Boolean value that indicates whether the information should be written into the registry even if a conflict is found with another driver or device. The default value is FALSE.

ConflictDetected

Points to a Boolean value set to TRUE on return from **IoReportResourceUsage** if a previously loaded driver has already claimed a resource specified in the caller's *DriverList* or *DeviceList*.

Include

ntddk.h

Return Value

IoReportResourceUsage can return one of the following:

STATUS_SUCCESS
STATUS_INSUFFICIENT_RESOURCES

Comments

The values supplied in the CM_RESOURCE_LIST must be identical to those found in the driver's call to **IoQueryDeviceDescription**, **HalGetBusDataByOffset**, or **HalGetBusData**, not those returned by the driver's calls to **HalTranslateBusAddress**, **HalGetAdapter**, or **HalGetInterruptVector**.

This routine automatically searches the configuration registry for resource conflicts between resources requested and resources claimed by previously installed drivers. The contents of *DriverList* or *DeviceList* are matched against all other resource lists stored in the registry to determine whether a conflict exists.

If no conflict is detected or if *OverrideConflict* is set to TRUE, this routine creates appropriate entries in the registry **ResourceMap** that contains the specified resource lists.

If *OverrideConflict* is set to FALSE, this routine logs an error recording the exact nature of the conflict that is displayed in the Win32 event viewer. If *OverrideConflict* is reset to TRUE, no such error is reported if a resource conflict exists and the caller's resource list is written into the registry. However, the caller cannot use any resource for which a conflict was detected.

If a driver claims resources on a device-specific basis for more than one device, the driver must call this routine for each such device.

This routine can be called more than once for a given device or driver. If a new resource list is given, it will replace the previous resource list in the registry. A driver must call **IoReportResourceUsage** with a *DriverList* or *DeviceList* CM_RESOURCE_LIST in which the **Count** is zero to erase its claim on resources in the registry if the driver is unloaded.

A CM_RESOURCE_LIST contains two variable-sized arrays. Each array has a default size of one. If either array has more than one element, memory must be allocated dynamically to contain the additional elements. A side effect of this definition is that only one CM_PARTIAL_RESOURCE_DESCRIPTOR can be part of each CM_FULL_RESOURCE_DESCRIPTOR in the list, except for the last full resource descriptor in the CM_RESOURCE_LIST, which can have additional partial resource descriptors in its array.

As an alternative, a device driver can call **HalGetBusDataByOffset** or **HalGetBusData** to locate its device(s) and to retrieve bus-relative configuration information. Then the driver can call **IoAssignResources**, which encapsulates most of the functionality of **IoReportResourceUsage**, allows the caller to specify preferred and alternative resources in a single IO_RESOURCE_REQUIREMENTS_LIST, and returns a CM_RESOURCE_LIST specifying the hardware resources it claimed on behalf of the caller. Drivers of PCI-type devices can call **HalAssignSlotResources**, rather than **IoReportResourceUsage** or **IoAssignResources**.

Callers of **IoReportResourceUsage** must be running at IRQL PASSIVE_LEVEL.

See Also

CM_FULL_RESOURCE_DESCRIPTOR, CM_PARTIAL_RESOURCE_DESCRIPTOR, CM_RESOURCE_LIST, **HalAssignSlotResources**, **HalGetBusData**, **HalGetBusDataByOffset**, **IoAssignResources**, **IoQueryDeviceDescription**

IoReportTargetDeviceChange

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoReportTargetDeviceChange*** in that book for a full reference.

IoReportTargetDeviceChangeAsynchronous

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoReportTargetDeviceChangeAsynchronous*** in that book for a full reference.

IoRequestDeviceEject

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoRequestDeviceEject*** in that book for a full reference.

IoRequestDpc

```
VOID
  IoRequestDpc(
  IN PDEVICE_OBJECT  DeviceObject,
  IN PIRP  Irp,
  IN PVOID  Context
  );
```

IoRequestDpc queues a driver-supplied DpcForIsr routine from the ISR to complete interrupt-driven I/O processing at a lower IRQL.

Parameters

DeviceObject

Poin270s to the device object for which the request that caused the interrupt is being processed.

Irp

Points to the current IRP for the specified device.

Context

Points to a driver-determined context to be passed to the DPC routine.

Include

wdm.h or *ntddk.h*

Comments

Callers of **IoRequestDpc** must be running at DIRQL.

Because **IoRequestDpc** is called from the device driver's ISR, the DIRQL is the *SynchronizeIrql* value that was specified when the driver called **IoConnectInterrupt**.

However, it is actually possible to queue a DPC at any IRQL >= DISPATCH_LEVEL using the **Ke..Dpc** routines.

See Also

IoInitializeDpcRequest, **KeInitializeDpc**, **KeInsertQueueDpc**

IoReuseIrp

```
VOID
  IoReuseIrp(
  IN OUT PIRP Irp,
  IN NTSTATUS Status
  );
```

IoReuseIrp reinitializes an IRP so that it can be reused.

Parameters

Irp

Points to the IRP to be reinitialized for reuse.

Status

Specifies the NTSTATUS value to be set in the IRP after it is reinitialized.

Include

ntddk.h

Comments

Drivers should call **IoReuseIrp**, and not **IoInitializeIrp**, to reinitialize an IRP.

A driver should use **IoReuseIrp** only on IRPs it previously allocated with **IoAllocateIrp**. It should not use this routine for IRPs created with **IoMakeAssociatedIrp**, **IoBuildSynchronousFsdRequest**, **IoBuildAsynchronousFsdRequest**, or **IoBuildDeviceIoControlRequest**.

Callers of **IoReuseIrp** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoInitializeIrp, **IoAllocateIrp**, **IoMakeAssociatedIrp**, IRP

IoSetCancelRoutine

```
PDRIVER_CANCEL
  IoSetCancelRoutine(
```

```
  IN PIRP  Irp,
  IN PDRIVER_CANCEL  CancelRoutine
  );
```

IoSetCancelRoutine sets up a driver-supplied Cancel routine to be called if a given IRP is canceled. This routine can disable the Cancel routine currently set in an IRP.

Parameters

Irp

Points to the IRP being put into or removed from a cancelable state.

CancelRoutine

Specifies the entry point of the caller-supplied Cancel routine to be called if the specified IRP is canceled or is NULL if the given IRP is being removed from the cancelable state. This routine is declared as follows:

```
VOID
(*PDRIVER_CANCEL)(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );
```

Include

wdm.h or *ntddk.h*

Return Value

IoSetCancelRoutine returns the previous value of **Irp->CancelRoutine**. If no Cancel routine was previously set, or if IRP cancellation is already in progress, **IoSetCancel-Routine** returns NULL.

Comments

A driver must hold the system cancel spin lock when calling this routine if the driver uses the I/O-manager-supplied device queue in the device object. The driver executes at IRQL DISPATCH_LEVEL after calling **IoAcquireCancelSpinLock** until it releases the cancel spin lock with **IoReleaseCancelSpinLock**.

If the driver manages its own queue(s) of IRPs, then the driver need not hold the cancel spin lock when calling this routine. **IoSetCancelSpinLock** uses an interlocked exchange intrinsic to set the address of the Cancel routine as an atomic operation. Reduced usage of the cancel spin lock can improve driver performance and overall system performance.

Driver Cancel routines are called at IRQL DISPATCH_LEVEL with the cancel spin lock held. The Cancel routine must release the cancel spin lock before it returns control.

See Also

IoAcquireCancelSpinLock, IoReleaseCancelSpinLock

IoSetCompletionRoutine

```
VOID
  IoSetCompletionRoutine(
  IN PIRP  Irp,
  IN PIO_COMPLETION_ROUTINE  CompletionRoutine,
  IN PVOID  Context,
  IN BOOLEAN  InvokeOnSuccess,
  IN BOOLEAN  InvokeOnError,
  IN BOOLEAN  InvokeOnCancel
  );
```

IoSetCompletionRoutine registers an IoCompletion routine to be called when the next-lower-level driver has completed the requested operation for the given IRP.

Parameters

Irp

Points to the IRP that the driver wants to track.

CompletionRoutine

Specifies the entry point for the driver-supplied IoCompletion routine to be called when the next-lower driver completes the packet. This routine is declared as follows:

```
NTSTATUS
(*PIO_COMPLETION_ROUTINE)(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN PVOID Context
    );
```

Context

Points to a driver-determined context to pass to the IoCompletion routine.

InvokeOnSuccess

Specifies whether the completion routine is called if the IRP is completed with STATUS_SUCCESS in the I/O status block.

InvokeOnError

Specifies whether the completion routine is called if the IRP is completed with an error STATUS_*XXX* in the I/O status block.

InvokeOnCancel

Specifies whether the completion routine is called if the IRP is completed with STATUS_CANCELLED set in the I/O status block.

Include

wdm.h or *ntddk.h*

Comments

This routine sets the transfer address of the IoCompletion routine in the given IRP. The lowest-level driver in a chain of layered drivers cannot call this routine.

IoSetCompletionRoutine registers the specified routine to be called when the next-lower-level driver has completed the requested operation in any or all of the following ways:

- Successfully
- With an error
- Canceled the IRP

Usually, the I/O status block is set by the underlying device driver. It is read but not altered by any higher-level drivers' IoCompletion routines.

Higher-level drivers that allocate IRP's with **IoAllocateIrp** or **IoBuildAsynchronous-FsdRequest** must call this routine with all *InvokeOnXxx* parameters set to TRUE before passing the driver-allocated IRP to **IoCallDriver**. When the IoCompletion routine is called with such an IRP, it must free the driver-allocated IRP and any other resources that the driver set up for the request, such as MDLs with **IoBuildPartialMdl**. Such a driver should return STATUS_MORE_PROCESSING_REQUIRED when it calls **IoFreeIrp** to forestall the I/O Manager's completion processing for the driver-allocated IRP.

Callers of **IoSetCompletionRoutine** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IO_STACK_LOCATION, **IoAllocateIrp**, **IoBuildAsynchronousFsdRequest**, IRP, **IoBuildPartialMdl**, **IoFreeIrp**

IoSetDeviceInterfaceState

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoSetDeviceInterfaceState*** in that book for a full reference.

IoSetHardErrorOrVerifyDevice

```
VOID
  IoSetHardErrorOrVerifyDevice(
  IN PIRP  Irp,
  IN PDEVICE_OBJECT  DeviceObject
  );
```

IoSetHardErrorOrVerifyDevice must be called before completing an IRP if a removable-media driver's call to **IoIsErrorUserInduced** returns TRUE.

Parameters

Irp

Points to the IRP for which the driver encountered a user-induced error.

DeviceObject

Points to the target device to be verified for the I/O operation.

Include

ntddk.h

Comments

A file system driver uses the corresponding file object of the specified device object to send a popup to the user who can correct the error and retry the operation or cancel it.

Calling **IoSetHardErrorOrVerifyDevice** ensures that the popup is sent to the user thread that originally made the I/O request to the target device on which the media might have changed.

Callers of **IoSetHardErrorOrVerifyDevice** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoIsErrorUserInduced, IoRaiseHardError, IoRaiseInformationalHardError

IoSetNextIrpStackLocation

```
VOID
  IoSetNextIrpStackLocation(
  IN OUT PIRP  Irp
  );
```

IoSetNextIrpStackLocation sets the IRP stack location in a driver-allocated IRP to that of the caller.

Parameters

Irp

Points to the IRP whose stack location is to be set.

Include

wdm.h or *ntddk.h*

Comments

In general, this routine is seldom used by drivers. It is primarily used by drivers that require their own stack location in an IRP that they have allocated, on their own, to send to another driver.

IoSetNextIrpStackLocation is generally not needed because either:

- The driver received the IRP it is passing from another, higher-level driver, and so it already owns a stack location,
- Or, the driver allocated the IRP but does not need its own stack location because it can keep everything that it needs in a context block whose address can be passed to its IoCompletion routine.

Care should be taken if this routine is called, especially when allocating the IRP with **IoAllocateIrp** or **IoMakeAssociatedIrp**. The writer of the allocating driver must remember that a caller-specific stack location is not included in the number of stack locations required by the lower-level drivers to which it sends IRPs with **IoCallDriver**. A driver must explicitly specify a stack location for itself in its call to **IoAllocateIrp** or **IoMakeAssociatedIrp** if it calls **IoSetNextIrpStackLocation** with the IRP returned by either routine.

A driver cannot call **IoSetNextIrpStackLocation** with any IRP it allocates by calling **IoBuildAsynchronousFsdRequest**, **IoBuildDeviceIoControlRequest**, or **IoBuildSynchronousFsdRequest**.

Callers of **IoSetNextIrpStackLocation** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IO_STACK_LOCATION, **IoAllocateIrp**, **IoBuildAsynchronousFsdRequest**, **IoBuildDeviceIoControlRequest**, **IoBuildSynchronousFsdRequest**, **IoCallDriver**, **IoSetCompletionRoutine**

IoSetPartitionInformation

```
NTSTATUS
  IoSetPartitionInformation(
```

```
IN PDEVICE_OBJECT  DeviceObject,
IN ULONG  SectorSize,
IN ULONG  PartitionNumber,
IN ULONG  PartitionType
);
```

IoSetPartitionInformation sets the partition type and number in a partition table entry for a given disk represented by the device object.

Parameters

DeviceObject

Points to the device object representing the device on which the partition type is to be set.

SectorSize

Specifies the size, in bytes, of sectors on the disk.

PartitionNumber

Specifies the partition number on the device whose partition type is to be set.

PartitionType

Specifies the type for the partition. For the currently defined *PartitionType* values, see the Win32 SDK.

Include

ntddk.h

Return Value

If **IoSetPartitionInformation** returns STATUS_SUCCESS, the disk driver updates its notion of the partition type for this partition in its device extension.

Comments

This routine is called when a disk device driver is asked to set the partition type in a partition table entry by an IRP_MJ_DEVICE_CONTROL request. This request is generally issued by the format utility, which performs I/O control functions on the partition. The driver passes a pointer to the device object representing the physical disk and the number of the partition associated with the device object that the format utility has open.

This routine is synchronous and must be called by the disk driver's Dispatch routine or by a driver thread. Thus, all user and file system threads must be prepared to enter a wait state when issuing the device control request to set the partition type for the device.

This routine assumes the partition number passed in by the disk driver actually exists.

This routine must be called at PASSIVE_LEVEL IRQL because it uses a kernel event object to synchronize I/O completion on the device. The event cannot be set to the Signaled state without queuing and executing the I/O system's special kernel APC routine for I/O completion.

See Also

IoReadPartitionTable, IoWritePartitionTable

IoSetShareAccess

```
VOID
  IoSetShareAccess(
  IN ACCESS_MASK  DesiredAccess,
  IN ULONG  DesiredShareAccess,
  IN OUT PFILE_OBJECT  FileObject,
  OUT PSHARE_ACCESS  ShareAccess
  );
```

IoSetShareAccess sets the access rights for sharing the given file object.

Parameters

DesiredAccess

Specifies the type of access requested for the *FileObject*. See **IoCreateFile** for a complete list of system-defined *DesiredAccess* flags.

DesiredShareAccess

Specifies the share access to be set for the file object. This value can be zero, one, or more of the following:

FILE_SHARE_READ
FILE_SHARE_WRITE
FILE_SHARE_DELETE

FileObject

Points to the file object whose share access is being set or reset.

ShareAccess

Points to the SHARE_ACCESS structure associated with *FileObject*. Drivers should treat this structure as opaque.

Include

wdm.h or *ntddk.h*

Comments

Only highest-level kernel-mode drivers should call this routine. The call must occur in the context of the first thread that attempts to open the *FileObject*.

This routine sets access and share access information when the *FileObject* is first opened. It returns a pointer to the common share-access data structure associated with *FileObject*. Callers should save this pointer for later use when updating the access or closing the file.

Generally, FSDs are most likely to call this routine. However, other highest-level drivers can call **IoSetShareAccess** to control the kind of access allowed to a driver-created device object associated with the given *FileObject*.

IoSetShareAccess is not an atomic operation. Therefore, drivers calling this routine must protect the shared file object passed to **IoSetShareAccess** by means of some kind of lock, such as a mutex or a resource lock, in order to prevent corruption of the shared access counts.

Callers of **IoSetShareAccess** must be running at IRQL PASSIVE_LEVEL.

See Also

IoCreateFile, **IoCheckShareAccess**, **IoGetFileObjectGenericMapping**, **IoGetRelatedDeviceObject**, **IoRemoveShareAccess**, **IoUpdateShareAccess**

IoSetThreadHardErrorMode

```
BOOLEAN
  IoSetThreadHardErrorMode(
  IN BOOLEAN  EnableHardErrors
  );
```

IoSetThreadHardErrorMode enables or disables hard error reporting for the current thread.

Parameters

EnableHardErrors

Specifies if hard error reporting to the user should be enabled or disabled for this thread.

Include

ntddk.h

Return Value

IoSetThreadHardErrorMode returns TRUE if hard errors were enabled from this thread before this routine completed execution.

Comments

If hard errors are disabled for a given thread, calls to **IoRaiseHardError** will not display a message to the user indicating that a serious error has occurred. In addition, the IRP that is passed to **IoRaiseHardError** is completed without any data being copied into user buffers. Calling **IoRaiseInformationalHardError** after disabling hard errors causes that routine to always return FALSE for this thread.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoRaiseHardError, **IoRaiseInformationalHardError**

IoSizeOfIrp

```
USHORT
  IoSizeOfIrp(
  IN CCHAR  StackSize
  );
```

IoSizeOfIrp determines the size in bytes for an IRP, given the number of stack locations in the IRP.

Parameters

StackSize

Specifies the number of stack locations for the IRP.

Include

wdm.h or *ntddk.h*

Return Value

IoSizeOfIrp returns the size, in bytes, of the IRP.

Comments

Callers of **IoSizeOfIrp** can be running at any IRQL level.

The input *StackSize* value is either that of the next-lower driver's device object or one more than that value.

See Also

IoAllocateIrp, **IoMakeAssociatedIrp**

IoSkipCurrentIrpStackLocation

```
VOID
  IoSkipCurrentIrpStackLocation(
  IN PIRP  Irp
  );
```

IoSkipCurrentIrpStackLocation copies the IRP stack parameters from the current stack location to the stack location of the next-lower driver and does not allow the current driver to set an I/O completion routine.

Parameters

Irp

Points to the IRP.

Include

wdm.h or *ntddk.h*

Comments

A driver calls **IoSkipCurrentIrpStackLocation** to copy the IRP parameters from its stack location to the next-lower driver's stack location. The caller of this routine does not set an I/O completion routine and thus is no longer involved in handling this IRP once it passes the IRP down the device stack with **IoCallDriver**.

A driver that calls this routine must not set an I/O completion routine for this IRP. Drivers that copy their IRP parameters and set a completion routine should call **IoCopyCurrentIrpStackLocationToNext** instead of this routine.

Callers of **IoSkipCurrentIrpStackLocation** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IO_STACK_LOCATION, **IoCallDriver**, **IoCopyCurrentIrpStackLocationToNext**

IoStartNextPacket

```
VOID
  IoStartNextPacket(
  IN PDEVICE_OBJECT  DeviceObject,
  IN BOOLEAN  Cancelable
  );
```

IoStartNextPacket dequeues the next IRP, if any, from the given device object's associated device queue and calls the driver's StartIo routine.

Parameters

DeviceObject

Points to the device object for which the IRP is to be dequeued.

Cancelable

Specifies whether IRPs in the device queue can be canceled.

Include

wdm.h or *ntddk.h*

Comments

If there are no IRPs currently in the device queue for the target *DeviceObject*, this routine simply returns control to the caller.

If the driver passed a pointer to a cancel routine when it called **IoStartPacket**, it should pass TRUE in the *Cancelable* parameter. If *Cancelable* is TRUE, the I/O Manager will use the cancel spin lock to protect the device queue and the current IRP.

Drivers that do not have a StartIo routine cannot call **IoStartNextPacket**.

Callers of **IoStartNextPacket** must be running at IRQL DISPATCH_LEVEL. Usually, this routine is called from a device driver's DpcForIsr or CustomDpc routine, both of which are run at IRQL DISPATCH_LEVEL.

See Also

DEVICE_OBJECT, **IoStartNextPacketByKey**, **IoStartPacket**

IoStartNextPacketByKey

```
VOID
  IoStartNextPacketByKey(
  IN PDEVICE_OBJECT  DeviceObject,
  IN BOOLEAN  Cancelable,
  IN ULONG  Key
  );
```

IoStartNextPacketByKey dequeues the next packet from the given device object's associated device queue according to a specified sort-key value and calls the driver's StartIo routine with that IRP.

Parameters

DeviceObject

Points to the device object for which the IRP is to be dequeued.

Cancelable

Specifies whether IRPs in the device queue can be canceled.

Key

Specifies the sort key that determines which entry to remove from the queue.

Include

wdm.h or *ntddk.h*

Comments

If there are no IRPs currently in the device queue for the target device object, this routine simply returns control to the caller.

If the driver passed a pointer to a cancel routine when it called **IoStartPacket**, it should pass TRUE in the *Cancelable* parameter. If *Cancelable* is TRUE, the I/O Manager will use the cancel spin lock to protect the device queue and the current IRP.

Drivers that do not have a StartIo routine cannot call **IoStartNextPacketByKey**.

Callers of **IoStartNextPacketByKey** must be running at IRQL <= DISPATCH_LEVEL. Usually, this routine is called from a device driver's DpcForIsr or CustomDpc routine, both of which are run at IRQL DISPATCH_LEVEL.

See Also

DEVICE_OBJECT, **IoStartNextPacket**, **IoStartPacket**

IoStartPacket

```
VOID
  IoStartPacket(
  IN PDEVICE_OBJECT  DeviceObject,
  IN PIRP  Irp,
  IN PULONG  Key  OPTIONAL,
  IN PDRIVER_CANCEL  CancelFunction  OPTIONAL
  );
```

IoStartPacket calls the driver's StartIo routine with the given IRP or inserts the IRP into the device queue associated with the given device object if the device is already busy.

Parameters

DeviceObject

Points to the target device object for the IRP.

Irp

Points to the IRP to be processed.

Key

Points to a value that determines where to insert the packet into the device queue. If this is zero, the packet is inserted at the tail of the device queue.

CancelFunction

Specifies the entry point for a driver-supplied Cancel routine.

Include

wdm.h or *ntddk.h*

Comments

If the driver is already busy processing a request for the target device object, then the packet is queued in the device queue. Otherwise, this routine calls the driver's StartIo routine with the specified IRP.

If a nonNULL *CancelFunction* pointer is supplied, it is set in the IRP so the driver's Cancel routine is called if the IRP is canceled before its completion.

Drivers that do not have a StartIo routine cannot call **IoStartPacket**.

Callers of **IoStartPacket** must be running at IRQL <= DISPATCH_LEVEL. Usually, this routine is called from a device driver's Dispatch routine at IRQL PASSIVE_LEVEL.

See Also

DEVICE_OBJECT, **IoMarkIrpPending**, **IoSetCancelRoutine**, **IoStartNextPacket**, **IoStartNextPacketByKey**

IoStartTimer

```
VOID
  IoStartTimer(
  IN PDEVICE_OBJECT  DeviceObject
  );
```

IoStartTimer enables the timer associated with a given device object so the driver-supplied IoTimer routine is called once per second.

Parameters

DeviceObject

Points to a device object whose timer routine is to be called.

Include

wdm.h or *ntddk.h*

Comments

The driver must already have set up the IoTimer routine for the *DeviceObject* by calling **IoInitializeTimer**.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoInitializeTimer, **IoStopTimer**, **KeInitializeDpc**, **KeInitializeTimer**, **KeSetTimer**

IoStopTimer

```
VOID
  IoStopTimer(
  IN PDEVICE_OBJECT  DeviceObject
  );
```

IoStopTimer disables the timer for a specified device object so the driver-supplied IoTimer routine is not called.

Parameters

DeviceObject

Points to the device object with which the IoTimer routine is associated.

Include

wdm.h or *ntddk.h*

Comments

The driver-supplied IoTimer routine can be re-enabled with a call to **IoStartTimer**.

Do not call **IoStopTimer** from within a driver's IoTimer routine.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoInitializeTimer, **IoStartTimer**

IoUnregisterPlugPlayNotification

This routine is documented in Volume 1 of the *Windows 2000 Driver Development Reference*. Please see ***IoUnregisterPlugPlayNotification*** in that book for a full reference.

IoUnregisterShutdownNotification

```
VOID
  IoUnregisterShutdownNotification(
  IN PDEVICE_OBJECT  DeviceObject
  );
```

IoUnregisterShutdownNotification removes a registered driver from the shutdown notification queue. The driver's DispatchShutdown routine is not called before the Power Manager sends an IRP_MN_SET_POWER request to shut down the system.

Parameters

DeviceObject

Points to the driver's device object.

Include

wdm.h or ntddk.h

Comments

IoUnregisterShutdownNotification can be called by a driver only if that driver previously called **IoRegisterShutdownNotification** with the given *DeviceObject*. This routine is usually called from a driver's Unload routine.

Calling **IoUnregisterShutdownNotification** cancels all shutdown notifications that have been registered for the given *DeviceObject*.Callers of **IoUnregisterShutdownNotification** must be running at IRQL PASSIVE_LEVEL.

See Also

IoRegisterShutdownNotification

IoUpdateShareAccess

```
VOID
  IoUpdateShareAccess(
  IN PFILE_OBJECT  FileObject,
  IN OUT PSHARE_ACCESS  ShareAccess
  );
```

IoUpdateShareAccess updates the share access for the given file object, usually when the file is being opened.

Parameters

FileObject

Points to a referenced file object representing the file or associated device object for which to update the share access.

ShareAccess

Points to the common SHARE_ACCESS structure associated with the *FileObject*. Drivers should treat this structure as opaque.

Include

ntddk.h

Comments

IoUpdateShareAccess is not an atomic operation. Therefore, drivers calling this routine must protect the shared file object passed to **IoUpdateShareAccess** by means of some kind of lock, such as a mutex or a resource lock, in order to prevent corruption of the shared access counts.

Before calling **IoUpdateShareAccess**, the caller must successfully call **IoCheckShare-Access** with *Update* set to False. Such a call to **IoCheckShareAccess** determines whether the requested shared access is compatible with the way the file object is currently being accessed by other opens, but it does not update the SHARE_ACCESS structure. **IoUpdate-ShareAccess** actually updates the SHARE_ACCESS structure associated with the file object.

Callers of **IoUpdateShareAccess** must be running at IRQL PASSIVE_LEVEL.

See Also

IoCheckShareAccess, **IoRemoveShareAccess**, **IoSetShareAccess**

IoWMIAllocateInstanceIds

```
NTSTATUS
  IoWMIAllocateInstanceIds(
  IN GUID  *Guid,
  IN ULONG  InstanceCount,
  OUT ULONG  *FirstInstanceId
  );
```

IoWMIAllocateInstanceIds allocates one or more instance IDs unique to the GUID.

Parameters

Guid

Points to the GUID for which to generate instance identifiers.

InstanceCount

Specifies how many instance identifiers should be provided.

FirstInstanceId

Points to the first instance identifier that the driver should use.

Include

wdm.h or *ntddk.h*

Return Value

IoWMIAllocateInstanceIds returns a status from the following list:

STATUS_SUCCESS

Indicates that WMI successfully provided unique instance identifiers for the GUID specified.

STATUS_UNSUCCESSFUL

Indicates that the WMI services are not available.

STATUS_INSUFFICIENT RESOURCES

Indicates that insufficient resources were available to provide the caller with instance IDs.

Comments

If greater than one instance was requested in *InstanceCount* and the routine completed successfully, *FirstInstanceId* points to the first instance that the caller should use. For each instance requested beyond one, the caller should increment the value returned in **First-InstanceId.* For example, if the caller requested six instances and one was returned as the

value of *FirstInstanceId*, the caller should use the values 1-6 as his unique instance identifiers.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

IoWMIDeviceObjectToProviderId

```
ULONG
  IoWMIDeviceObjectToProviderId(
  IN PDEVICEOBJECT  DeviceObject
  );
```

IoWMIDeviceObjectToProviderId translates the specified device object into the corresponding WMI Provider Id.

Parameters

DeviceObject

Points to a device object.

Include

wdm.h or *ntddk.h*

Return Value

IoWMIDeviceObjectToProviderId returns the WMI Provider Id associated with the specified device object.

Comments

IoWMIDeviceObjectToProviderId should be used when filling in the *ProviderId* member of the WNODE_HEADER structure in those cases when the WNODEHEADER structure is being initialized as part of a WNODE_EVENT_ITEM or WNODE_EVENT_REFERENCE structure. (If the WNODE_HEADER is being used for other purposes, *ProviderId* is reserved.)

When running on a 32 bit OS, the provider Id and the device object are identical. When running on a 64 bit OS, **IoWMIDeviceObjectToProviderId** will convert the 64 bit device object to a 32 bit provider ID.

See Also

WNODE_HEADER, WNODE_EVENT_ITEM, WNODE_EVENT_REFERENCE

IoWMIRegistrationControl

```
NTSTATUS
  IoWMIRegistrationControl(
  IN PDEVICE_OBJECT  DeviceObject,
  IN ULONG  Action
  );
```

IoWMIRegistrationControl is used to request WMI action, such as adding or removing a device object from WMI.

Parameters

DeviceObject

Points to either a device object or the address of a callback function. If the WMIREG_FLAG_CALLBACK is set in the *Action* parameter, then *DeviceObject* should contain a callback address. Otherwise, *DeviceObject* contains a device object pointer.

Action

Specifies the action that WMI should take based on the specified value:

Action	Action to be taken
WMIREG_ACTION_REGISTER	Specifies that WMI should register *DeviceObject* as a provider of WMI information.
WMIREG_ACTION_DEREGISTER	Specifies that WMI should remove *DeviceObject* from its list of WMI providers.
WMI_ACTION_REREGISTER	Specifies that WMI should unregister the device and then register (reregister) the device. Reregistering the device will result in WMI sending an IRP_MN_REGINFO to the device.
WMIREG_ACTION_UPDATE_GUIDS	Specifies that WMI should requery the device object for a new list of GUID identifiers that it provides data for. This will result in WMI sending an IRP_MN_REGINFO to the device.
WMIREG_FLAG_CALLBACK	Indicates that the value contained at *DeviceObject* is actually the address of a callback function. (Note: WMIREG_FLAG_CALLBACK is not supported in the Windows 98 implementation of WMI.)

Include

wdm.h or *ntddk.h*

Return Value

IoWMIRegistrationControl returns a status code from the following list:

STATUS_SUCCESS

Indicates that WMI completed the action requested without error.

STATUS_INVALID_PARAMETER

Indicates that the action, specified in *Action*, was invalid.

STATUS_Xxx

Indicates that the request failed for the reason specified by the NTSTATUS value. See *ntstatus.h* for detailed information for the actual status return code.

Comments

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

IoWMISuggestInstanceName

```
NTSTATUS
  IoWMISuggestInstanceName(
  IN PDEVICE_OBJECT DeviceObject OPTIONAL,
  IN PUNICODE_STRING SymbolicLinkName OPTIONAL,
  IN BOOLEAN CombineNames,
  OUT PUNICODE_STRING SuggestedInstanceName
  );
```

IoWmiSuggestInstanceName is used to request that WMI suggest a base name that a driver can use to build WMI instance names for the device.

Parameters

PhyiscalDeviceObject

If supplied, points to the driver's physical device object.

SymbolicLinkName

If supplied, points to the symbolic link name returned from *IoRegisterDeviceInterface*.

CombineNames

If TRUE then the suggested names returned will combine the Physical*DeviceObject* and *SymbolicLinkName* information.

SuggestedInstanceName

A pointer to a buffer which upon successful completion will contain a UNICODE STRING that contains the suggested instance name. The caller is responsible for freeing this buffer when it is no longer needed.

Include

wdm.h or *ntddk.h*

Return Value

IoWMISuggestInstanceName returns a status code from the following list:

STATUS_SUCCESS

Indicates that WMI was able to successfully complete this function.

STATUS_UNSUCCESSFUL

Indicates that the WMI services are not available.

STATUS_INSUFFICIENT_RESOURCES

Indicates that insufficient resources were available to provide the caller with a buffer containing the UNICODE string.

STATUS_NO_MEMORY

Indicates that insufficient resources were available to provide the caller with a buffer containing the UNICODE string.

Comments

If CombineNames is TRUE then both PhysicalDeviceObject and SymbolicLinkName must be specified. Otherwise only one of them should be specified.

IoWMIWriteEvent

```
NTSTATUS
  IoWMIWriteEvent(
  IN PVOID WnodeEventItem
  );
```

IoWMIWriteEvent delivers a given event to the user mode WMI components for notification.

Parameters

WnodeEventItem

Points to a WNODE_EVENT_ITEM structure to be delivered to the user mode WMI components that requested notification of the event.

Include

wdm.h or *ntddk.h*

Return Value

IoWMIWriteEvent returns a status code from the following list:

STATUS_SUCCESS

Indicates that WMI has successfully queued the event for delivery to the user mode WMI components.

STATUS_UNSUCCESSFUL

Indicates that WMI services are unavailable.

STATUS_BUFFER_OVERFLOW

Indicates that the event item specified exceeds the maximum allowed size.

STATUS_INSUFFICENT_RESOURCES

Indicates that insufficient resources were available for WMI to queue the event for delivery.

Comments

The WNODE_EVENT_ITEM structure that is allocated by the caller and passed in *WnodeEventItem* must be allocated from nonpaged pool. If **IoWMIWriteEvent** returns STATUS_SUCCESS, the memory for the event item will automatically be freed by the system. If **IOWMIWriteEvent** returns anything other than STATUS_SUCCESS, it is the caller's responsibility to free the buffer.

Drivers should only call **IoWMIWriteEvent** for events that have been enabled for WMI. This ensures that there is an event consumer waiting for indication on that event.

Callers of this routine must be running at IRQL < DISPATCH_LEVEL.

See Also

WNODE_EVENT_ITEM, *IOWmiDeviceObjectToProviderId*

IoWriteErrorLogEntry

```
VOID
  IoWriteErrorLogEntry(
  IN PVOID  ElEntry
  );
```

IoWriteErrorLogEntry queues a given error log packet to the system error logging thread.

Parameters

ElEntry

Points to the error log packet the driver has allocated with **IoAllocateErrorLogEntry** and filled in.

Include

wdm.h or *ntddk.h*

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateErrorLogEntry

IoWritePartitionTable

```
NTSTATUS
  IoWritePartitionTable(
  IN PDEVICE_OBJECT  DeviceObject,
  IN ULONG  SectorSize,
  IN ULONG  SectorsPerTrack,
  IN ULONG  NumberOfHeads,
  IN struct _DRIVE_LAYOUT_INFORMATION  *PartitionBuffer
  );
```

IoWritePartitionTable writes partition tables from the entries in the partition list buffer for each partition on the disk represented by the given device object.

Parameters

DeviceObject

Points to the device object representing the disk whose partition tables are to be written.

SectorSize

Specifies the size in bytes of sectors on the device.

SectorsPerTrack

Specifies the track size on the device.

NumberOfHeads

Specifies the number of tracks per cylinder.

PartitionBuffer

Points to the drive layout buffer that contains the partition list entries. For more detailed information about the DRIVE_LAYOUT_INFORMATION structure, see **IoReadPartitionTable**.

Include

ntddk.h

Return Value

If all writes are completed without error, **IoWritePartitionTable** returns STATUS_SUCCESS.

Comments

IoWritePartitionTable is called when a disk device driver is asked to set the partition type in a partition table entry or to repartition the disk by an IRP_MJ_DEVICE_CONTROL request. The device control request is generally issued by the format utility, which performs I/O control functions on the partitions and disks in the machine.

To reset a partition type, the driver passes a pointer to the device object representing the physical disk and the number of the partition associated with the device object that the format utility has open. When a disk is to be repartitioned dynamically, the disk driver must tear down its set of device objects representing the current disk partitions and create a new set of device objects representing the new partitions on the disk.

Applications that create and delete partitions and require full descriptions of the system should call **IoReadPartitionTable** with *ReturnRecognizedPartitions* set to FALSE. The drive layout structure can be modified by the system format utility to reflect a new configuration of the disk.

IoWritePartitionTable is synchronous. It must be called by the disk driver's Dispatch routine or by a driver thread. Thus, all user and file system threads must be prepared to enter a wait state when issuing the device control request to reset partition types for the device.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

IoCreateDevice, IoReadPartitionTable, IoSetPartitionInformation

CHAPTER 5

Kernel Routines

All kernel-mode drivers except video and SCSI miniport drivers and NDIS drivers are likely to call at least some **Ke***Xxx* routines.

References for the **Ke***Xxx* routines are in alphabetical order.

For an overview of the functionality of these routines, see Chapter 1, *Summary of Kernel-Mode Support Routines*.

KeAcquireSpinLock

```
VOID
  KeAcquireSpinLock(
  IN PKSPIN_LOCK  SpinLock,
  OUT PKIRQL  OldIrql
  );
```

KeAcquireSpinLock acquires a spin lock so the caller can synchronize access to shared data in a multiprocessor-safe way by raising IRQL.

Parameters

SpinLock

Pointer to an initialized spin lock for which the caller provides the storage.

OldIrql

Pointer to a variable that is set to the current IRQL when this call occurs.

Include

wdm.h or *ntddk.h*

Comments

The current IRQL is saved in *OldIrql*. Then, the current IRQL is reset to DISPATCH_LEVEL, and the specified spin lock is acquired.

The *OldIrql* value must be specified when the spin lock is released with **KeReleaseSpinLock**.

Spin locks can cause serious problems if not used judiciously. In particular, no deadlock protection is performed and dispatching is disabled while the spin lock is held. Therefore:

- The code within a critical region guarded by an spin lock must neither be pageable nor make any references to pageable data.
- The code within a critical region guarded by a spin lock can neither call any external function that might access pageable data or raise an exception, nor can it generate any exceptions.
- The caller should release the spin lock with **KeReleaseSpinLock** as quickly as possible.

Callers of **KeAcquireSpinLock** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeAcquireSpinLockAtDpcLevel, **KeInitializeSpinLock**, **KeReleaseSpinLock**

KeAcquireSpinLockAtDpcLevel

```
VOID
  KeAcquireSpinLockAtDpcLevel(
  IN PKSPIN_LOCK  SpinLock
  );
```

KeAcquireSpinLockAtDpcLevel acquires a spin lock when the caller is already running at IRQL DISPATCH_LEVEL.

Parameters

SpinLock

Pointer to an initialized spin lock for which the caller must provide the storage.

Include

wdm.h or *ntddk.h*

Comments

Drivers call **KeAcquireSpinLockAtDpcLevel** instead of **KeAcquireSpinLock** for better driver performance if and only if they are already running at IRQL DISPATCH_LEVEL.

If a driver is running at IRQL < DISPATCH_LEVEL, it should call **KeAcquireSpinLock** to have IRQL raised by that routine. **KeAcquireSpinLockAtDpcLevel** assumes the caller is already running at IRQL DISPATCH_LEVEL, so no raise is necessary.

The caller should release the spin lock with **KeReleaseSpinLockFromDpcLevel** as quickly as possible.

See Also

KeAcquireSpinLock, **KeInitializeSpinLock**, **KeReleaseSpinLock**, **KeReleaseSpinLockFromDpcLevel**

KeBugCheck

```
VOID
  KeBugCheck(
  IN ULONG  BugCheckCode
  );
```

KeBugCheck brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that would corrupt the system if the caller continued to run.

Parameters

BugCheckCode

Specifies a value that indicates the reason for the bug check.

Include

ntddk.h

Comments

A bug check is a system-detected error that causes an immediate, controlled shutdown of the system. Various kernel-mode components perform run-time consistency checking. When such a component discovers an unrecoverable inconsistency, it causes a bug check to be generated.

Whenever possible, all kernel-mode components should log an error and continue to run, rather than calling **KeBugCheck**. For example, if a driver is unable to allocate required resources, it should log an error so that the system continues to run; it must not generate a bug check. A driver or other kernel-mode component should call this routine only in cases of a fatal, unrecoverable error that could corrupt the system itself.

When a bug check is unavoidable, most system components call **KeBugCheckEx**, which provides more information about the cause of such an inconsistency than **KeBugCheck**.

Callers of **KeBugCheck** can be running at any IRQL.

See Also

IoAllocateErrorLogEntry, IoWriteErrorLogEntry, KeBugCheckEx, KeRegisterBugCheckCallback

KeBugCheckEx

```
VOID
  KeBugCheckEx(
  IN ULONG      BugCheckCode,
  IN ULONG_PTR  BugCheckParameter1,
  IN ULONG_PTR  BugCheckParameter2,
  IN ULONG_PTR  BugCheckParameter3,
  IN ULONG_PTR  BugCheckParameter4
  );
```

KeBugCheckEx brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that would corrupt the system if the caller continued to run.

Parameters

BugCheckCode

Specifies a value that indicates the reason for the bug check.

BugCheckParameterX

Supply additional information, such as the address and data where a memory-corruption error occurred, depending on the value of *BugCheckCode*.

Include

wdm.h or *ntddk.h*

Comments

A bug check is a system-detected error that causes an immediate, controlled shutdown of the system. Various kernel-mode components perform run-time consistency checking. When such a component discovers an unrecoverable inconsistency, it causes a bug check to be generated.

Whenever possible, all kernel-mode components should log an error and continue to run, rather than calling **KeBugCheck**. For example, if a driver is unable to allocate required resources, it should log an error so that the system continues to run; it must not generate a bug check. A driver or other kernel-mode component should call this routine only in cases of a fatal, unrecoverable error that could corrupt the system itself.

Callers of **KeBugCheckEx** can be running at any IRQL.

See Also

IoAllocateErrorLogEntry, **IoWriteErrorLogEntry**, **KeBugCheck**, **KeRegisterBugCheckCallback**

KeCancelTimer

```
BOOLEAN
  KeCancelTimer(
  IN PKTIMER  Timer
  );
```

KeCancelTimer dequeues a timer object before the timer interval, if any was set, expires.

Parameters

Timer

Pointer to an initialized timer object, for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Return Value

If the specified timer object is in the system timer queue, **KeCancelTimer** returns TRUE.

Comments

If the timer object is currently in the system timer queue, it is removed from the queue. If a DPC object is associated with the timer, it too is canceled. Otherwise, no operation is performed.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeInitializeTimer, **KeReadStateTimer**, **KeSetTimer**

KeClearEvent

```
VOID
  KeClearEvent(
  IN PRKEVENT Event
  );
```

KeClearEvent sets the given event to a not signaled state.

Parameters

Event

Pointer to an initialized dispatcher object of type event for which the caller supplies the storage.

Include

wdm.h or *ntddk.h*

Comments

Event is set to a not signaled state, meaning its value is set to zero.

For better performance, use **KeClearEvent** unless the caller uses the value returned by **KeResetEvent** to determine what to do next.

Callers of **KeClearEvent** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeInitializeEvent, **KeReadStateEvent**, **KeResetEvent**, **KeSetEvent**

KeDelayExecutionThread

```
NTSTATUS
  KeDelayExecutionThread(
  IN KPROCESSOR_MODE  WaitMode,
  IN BOOLEAN  Alertable,
  IN PLARGE_INTEGER  Interval
  );
```

KeDelayExecutionThread routine puts the current thread into an alertable or nonalertable wait state for a given interval.

Parameters

WaitMode

Specifies the processor mode in which the caller is waiting, which can be either **KernelMode** or **UserMode**. Lower-level drivers should specify **KernelMode**.

Alertable

Specifies TRUE if the wait is alertable. Lower-level drivers should specify FALSE.

Interval

Specifies the absolute or relative time, in units of 100 nanoseconds, for which the wait is to occur. A negative value indicates relative time. Absolute expiration times track any changes in system time; relative expiration times are not affected by system time changes.

Include

wdm.h or *ntddk.h*

Return Value

KeDelayExecutionThread returns one of the following values that describes how the delay was completed:

STATUS_SUCCESS

The delay completed because the specified interval elapsed.

STATUS_ALERTED

The delay completed because the thread was alerted.

STATUS_USER_APC

A user-mode APC was delivered before the given *Interval* expired.

Comments

The expiration time is computed and the current thread is put in a wait state. When the specified interval has passed, the thread exits the wait state and is put in the ready state, becoming eligible for execution.

The *Alertable* parameter specifies whether the thread can be alerted and its wait state consequently aborted. If the value of this parameter is FALSE then the thread cannot be alerted, no matter what the value of the *WaitMode* parameter or the origin of the alert. The only exception to this rule is that of a terminating thread. A thread is automatically made alertable, for instance, when terminated by a user with a CTRL+C.

If the value of *Alertable* is TRUE and one of the following conditions is present, the thread will be alerted:

1. If the origin of the alert is an internal, undocumented kernel-mode routine used to alert threads.
2. The origin of the alert is a user-mode APC, and the value of the *WaitMode* parameter is **UserMode**.

In the first of these two cases, the thread's wait is satisfied with a completion status of STATUS_ALERTED; in the second case, it is satisfied with a completion status of STATUS_USER_APC.

The thread must be alertable for a user-mode APC to be delivered. This is not the case for kernel-mode APCs. A kernel-mode APC can be delivered and executed even though the thread is not alerted. Once the APC's execution completes, the thread's wait resumes. A thread is never alerted, nor is its wait aborted, by the delivery of a kernel-mode APC.

The delivery of kernel-mode APCs to a thread that has called **KeDelayExecutionThread** does not depend on whether the thread can be alerted. If the kernel-mode APC is a special kernel-mode APC, then the APC is delivered provided that IRQL < APC_LEVEL. If the kernel-mode APC is a normal kernel-mode APC, then the APC is delivered provided that the following three conditions hold:

1. IRQL < APC_LEVEL.
2. No kernel-mode APC is in progress.
3. The thread is not in a critical section.

If the *WaitMode* parameter is **UserMode**, the kernel stack can be swapped out during the wait. Consequently, a caller must *never* attempt to pass parameters on the stack when calling **KeDelayExecutionThread** using the **UserMode** argument.

It is especially important to check the return value of **KeDelayExecutionThread** when the *WaitMode* parameter is **UserMode** or *Alertable* is TRUE, because **KeDelayExecution-Thread** might return early with a status of STATUS_USER_APC or STATUS_ALERTED.

All long term waits that can be aborted by a user should be **UserMode** waits and *Alertable* should be set to FALSE.

Where possible, *Alertable* should be set to FALSE and *WaitMode* should be set to **Kernel-Mode**, in order to reduce driver complexity. The principal exception to this is when the wait is a long term wait.

The expiration time of the delay is expressed as either an absolute time at which the delay is to expire, or a time relative to the current system time. If the *Interval* parameter is a negative value, the expiration time is relative.

Callers of **KeDelayExecutionThread** must be running at IRQL = PASSIVE_LEVEL.

See Also

KeQuerySystemTime

KeDeregisterBugCheckCallback

```
BOOLEAN
  KeDeregisterBugCheckCallback(
  IN PKBUGCHECK_CALLBACK_RECORD  CallbackRecord
  );
```

KeDeregisterBugCheckCallback removes a device driver's callback routine from the set of registered bug-check callbacks.

Parameters

CallbackRecord

Pointer to the caller-allocated storage containing an initialized bug-check callback record. This pointer was previously passed in successful calls to **KeInitializeCallbackRecord** and **KeRegisterBugCheckCallback**.

Include

ntddk.h

Return Value

KeDeregisterBugCheckCallback returns TRUE if the caller-supplied bug-check callback routine will no longer be called if a bug check occurs. If the given callback record was not registered, it returns FALSE.

Comments

KeDeregisterBugCheckCallback is the reciprocal of **KeRegisterBugCheckCallback**.

If an unloadable device driver sets up a bug-check callback routine with **KeRegisterBugCheckCallback**, that driver's Unload routine must call **KeDeregisterBugCheckCallback** before it frees the storage it allocated at *CallbackRecord*.

Callers of **KeDeregisterBugCheckCallback** can be running at any IRQL. Usually, a device driver is running at IRQL PASSIVE_LEVEL in its Unload routine when it calls **KeDeregisterBugCheckCallback**.

See Also

KeBugCheck, **KeBugCheckEx**, **KeInitializeCallbackRecord**, **KeRegisterBugCheckCallback**

KeEnterCriticalRegion

```
VOID
  KeEnterCriticalRegion( );
```

KeEnterCriticalRegion disables the delivery of normal kernel APCs temporarily. Special kernel-mode APCs are still delivered.

Include

ntddk.h

Comments

Highest-level drivers can call this routine while running in the context of the thread that requested the current I/O operation. Any caller of this routine should call **KeLeaveCriticalRegion** as quickly as possible.

Callers of **KeEnterCriticalRegion** must be running at IRQL PASSIVE_LEVEL.

See Also

KeLeaveCriticalRegion

KeFlushIoBuffers

```
VOID
  KeFlushIoBuffers(
  IN PMDL  Mdl,
  IN BOOLEAN  ReadOperation,
  IN BOOLEAN  DmaOperation
  );
```

KeFlushIoBuffers flushes the memory region described by an MDL from caches of all processors.

Parameters

Mdl

Pointer to an MDL that describes the range for the I/O buffer.

ReadOperation

Specifies TRUE if the flush is being performed for a read operation.

DmaOperation

Specifies TRUE for a DMA transfer, FALSE for PIO.

Include

wdm.h or *ntddk.h*

Comments

Drivers call **KeFlushIoBuffers** to maintain data integrity during DMA or PIO device transfer operations. Calling this routine affects all processors in the machine.

If *ReadOperation* is TRUE, the driver is reading information from the device to system memory, so valid data still might be in the processor instruction and data caches. **KeFlushIoBuffers** flushes data from all processors' caches to system memory, including the processor on which the caller is running.

If *ReadOperation* is FALSE, the driver is writing data from system memory to a device, so valid data might be in the processor's data cache but not yet transferred to the device. **KeFlushIoBuffers** flushes all processor's data caches, including the processor on which the caller is running.

As a general rule, drivers should call this routine just before beginning a DMA transfer operation or immediately following any PIO read operation.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

FlushAdapterBuffers

KeGetCurrentIrql

```
KIRQL
  KeGetCurrentIrql( );
```

KeGetCurrentIrql returns the current IRQL.

Include

wdm.h or *ntddk.h*

See Also

KeAcquireSpinLockAtDpcLevel, **KeLowerIrql**, **KeRaiseIrql**

KeGetCurrentProcessorNumber

```
ULONG
  KeGetCurrentProcessorNumber( );
```

KeGetCurrentProcessorNumber returns the system-assigned number of the current processor on which the caller is running.

Include

ntddk.h

Comments

KeGetCurrentProcessorNumber can be called by NT drivers to debug spin lock usage on SMP machines during driver development. An NT driver also might call **KeGetCurrentProcessorNumber** if it maintained some per-processor data and attempted to reduce cache-line contention.

The number of processors in an SMP machine is a zero-based value.

If the call to **KeGetCurrentProcessorNumber** occurs at IRQL < DISPATCH_LEVEL, a processor switch can occur between instructions. Consequently, callers of **KeGetCurrentProcessorNumber** usually run at IRQL >= DISPATCH_LEVEL.

KeGetCurrentThread

```
PRKTHREAD
  KeGetCurrentThread( );
```

KeGetCurrentThread identifies the current thread.

Return Value

KeGetCurrentThread returns a pointer to an opaque thread object.

Include

wdm.h or *ntddk.h*

Comments

A caller of **KeGetCurrentThread** can use the returned pointer as an input parameter to **KeQueryPriorityThread**, **KeSetBasePriorityThread**, or **KeSetPriorityThread**. However, the memory containing the thread object is opaque, that is, reserved for exclusive use by the system.

Callers of **KeGetCurrentThread** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeQueryPriorityThread, **KeSetBasePriorityThread**, **KeSetPriorityThread**, **PsGetCurrentThread**

KeGetDcacheFillSize

```
ULONG
  KeGetDcacheFillSize( );
```

KeGetDcacheFillSize is obsolete. Drivers should call **GetDmaAlignment** instead.

KeInitializeCallbackRecord

```
VOID
  KeInitializeCallbackRecord(
  IN PKBUGCHECK_CALLBACK_RECORD  CallbackRecord
  );
```

Device drivers call **KeInitializeCallbackRecord** to initialize a bug-check callback record before calling **KeRegisterBugCheckCallback**.

Parameters

CallbackRecord

Pointer to a caller-allocated nonpaged buffer, which must be at least **sizeof**(KBUGCHECK_CALLBACK_RECORD).

Include

ntddk.h

Comments

Before a device driver calls **KeRegisterBugCheckCallback**, it must call **KeInitializeCallbackRecord**.

Such a driver must provide resident storage for a bug-check record, which can be in a device extension, in a controller extension, in nonpaged pool allocated by the driver, or statically allocated in the driver. The structure and contents of the memory at *CallbackRecord* should be considered opaque, but this record must be preserved unless the driver has called **KeDeregisterBugCheckCallback**. After this call, the device driver is responsible for freeing the memory it allocated for the bug-check record if necessary.

Callers of **KeInitializeCallbackRecord** can be running at any IRQL. Usually, a device driver is running at IRQL PASSIVE_LEVEL in its **DriverEntry** routine when it calls **KeInitializeCallbackRecord**.

See Also

ExAllocatePool, **KeDeregisterBugCheckCallback**, **KeRegisterBugCheckCallback**

KeInitializeDeviceQueue

```
VOID
  KeInitializeDeviceQueue(
  IN PKDEVICE_QUEUE  DeviceQueue
  );
```

KeInitializeDeviceQueue initializes a device queue object to a not busy state.

Parameters

DeviceQueue

Pointer to a device queue object for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Comments

KeInitializeDeviceQueue initializes the specified device queue and sets its state to not busy.

A driver should call **KeInitializeDeviceQueue** from its AddDevice routine after creating the device object for the associated device. Storage for the device queue object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller.

See Also

KeInsertByKeyDeviceQueue, **KeInsertDeviceQueue**, **KeRemoveDeviceQueue**, **KeRemoveEntryDeviceQueue**

KeInitializeDpc

```
VOID
  KeInitializeDpc(
  IN PRKDPC  Dpc,
  IN PKDEFERRED_ROUTINE  DeferredRoutine,
  IN PVOID  DeferredContext
  );
```

KeInitializeDpc initializes a DPC object, setting up a deferred procedure that can be called with a given context.

Parameters

Dpc

Pointer to a DPC object for which the caller provides the storage.

DeferredRoutine

Specifies the entry point for a routine to be called when the DPC object is removed from the DPC queue. A *DeferredRoutine* is declared as follows:

```
VOID
(*PKDEFERRED_ROUTINE)(
    IN PKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1;
    IN PVOID SystemArgument2
    );
```

DeferredContext

Pointer to a caller-supplied context to be passed to the *DeferredRoutine* when it is called.

Include

wdm.h or *ntddk.h*

Comments

The caller can queue an initialized DPC with **KeInsertQueueDpc**. The caller also can set up a timer object associated with the initialized DPC object and queue the DPC with **KeSetTimer**.

Storage for the DPC object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in non-paged pool allocated by the caller.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

KeInsertQueueDpc, **KeRemoveQueueDpc**, **KeSetTimer**

KeInitializeEvent

```
VOID
  KeInitializeEvent(
  IN PRKEVENT  Event,
  IN EVENT_TYPE  Type,
  IN BOOLEAN  State
  );
```

KeInitializeEvent initializes an event object as a synchronization (single waiter) or notification type event and sets it to a signaled or not signaled state.

Parameters

Event

Pointer to an event object, for which the caller provides the storage.

Type

Specifies the event type, either **NotificationEvent** or **SynchronizationEvent**.

State

Specifies the initial state of the event. TRUE indicates a signaled state.

Include

wdm.h or *ntddk.h*

Comments

A caller cannot wait at raised IRQL for a nonzero interval on an event object or in a non-arbitrary thread context.

Storage for an event object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in non-paged pool allocated by the caller. If you allocate the event on the stack, you must specify a **KernelMode** wait when calling **KeWaitForSingleObject**, **KeWaitForMutexObject**, or **KeWaitForMultipleObjects**. During a **KernelMode** wait, the stack containing the event will not be paged out.

Drivers typically use a **NotificationEvent** to wait for an I/O operation to complete. When a notification event is set to the signaled state, all threads that were waiting on the event become eligible for execution. The event remains in the signaled state until a thread calls **KeResetEvent** or **KeClearEvent** to set the event in the not-signaled state.

A **SynchronizationEvent** is also called an *autoreset* or *autoclearing* event. When such an event is set, a single waiting thread becomes eligible for execution. The Kernel automatically resets the event to the not-signaled state each time a wait is satisfied. A driver might use a synchronization event to protect a shared resource that is used in synchronizing the operations of several threads. Synchronization events are rarely used in a typical driver.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

KeClearEvent, **KeReadStateEvent**, **KeResetEvent**, **KeSetEvent**, **KeWaitForMultipleObjects**, **KeWaitForSingleObject**

KeInitializeMutex

```
VOID
  KeInitializeMutex(
  IN PRKMUTEX  Mutex,
  IN ULONG  Level
  );
```

KeInitializeMutex initializes a mutex object at a given level number, setting it to a signaled state.

Parameters

Mutex

Pointer to a mutex object, for which the caller provides the storage.

Level

Specifies the level number to be assigned to the mutex.

Include

wdm.h or *ntddk.h*

Comments

For better performance, use the **Ex..FastMutex** routines instead of the **Ke..Mutex**. However, a fast mutex cannot be acquired recursively, as a kernel mutex can.

The mutex object is initialized with the specified *Level* and an initial state of signaled.

A driver cannot wait at raised IRQL nor in an arbitrary thread context for a nonzero interval on a mutex object.

Storage for a mutex object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

ExInitializeFastMutex, **KeReadStateMutex**, **KeReleaseMutex**, **KeWaitForMultipleObjects**, **KeWaitForMutexObject**, **KeWaitForSingleObject**

KeInitializeSemaphore

```
VOID
  KeInitializeSemaphore(
  IN PRKSEMAPHORE  Semaphore,
  IN LONG  Count,
  IN LONG  Limit
  );
```

KeInitializeSemaphore initializes a semaphore object with a given count and specifies an upper limit that the count can attain.

Parameters

Semaphore

Pointer to a dispatcher object of type semaphore, for which the caller provides the storage.

Count

Specifies the initial count value to be assigned to the semaphore. This value must be positive. A nonzero value sets the initial state of the semaphore to signaled.

Limit

Specifies the maximum count value that the semaphore can attain. This value must be positive. It determines how many waiting threads become eligible for execution when the semaphore is set to the signaled state and can therefore access the resource that the semaphore protects.

Include

wdm.h or *ntddk.h*

Comments

The semaphore object is initialized with the specified initial count and limit.

Storage for a semaphore object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the caller.

Callers of **KeInitializeSemaphore** must be running at IRQL PASSIVE_LEVEL.

See Also

KeReadStateSemaphore, **KeReleaseSemaphore**, **KeWaitForMultipleObjects**, **KeWaitForSingleObject**

KeInitializeSpinLock

```
VOID
  KeInitializeSpinLock(
  IN PKSPIN_LOCK  SpinLock
  );
```

KeInitializeSpinLock initializes a variable of type KSPIN_LOCK.

Parameters

SpinLock

Pointer to a spin lock, for which the caller must provide the storage.

Include

wdm.h or *ntddk.h*

Comments

This routine must be called before an initial call to **KeAcquireSpinLock** or to any other support routine that requires a spin lock as an argument.

Storage for a spin lock object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in non-paged pool allocated by the caller.

Callers of this routine can be running at any IRQL. Usually, a caller is running at IRQL PASSIVE_LEVEL in an AddDevice routine.

See Also

KeAcquireSpinLock, **KeAcquireSpinLockAtDpcLevel**, **KeReleaseSpinLock**

KeInitializeTimer

```
VOID
  KeInitializeTimer(
  IN PKTIMER  Timer
  );
```

KeInitializeTimer initializes a timer object.

Parameters

Timer

Pointer to a timer object, for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Comments

The timer object is initialized to a not signaled state.

Storage for a timer object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in non-paged pool allocated by the caller.

KeInitializeTimer can only initialize a notification timer. Use **KeInitializeTimerEx** to initialize a notification timer or a synchronization timer.

Callers of **KeInitializeTimer** should be running at IRQL DISPATCH_LEVEL or lower. It is best to initialize timers at IRQL PASSIVE_LEVEL.

Use **KeSetTimer** or **KeSetTimerEx** to define when the timer will expire.

See Also

KeCancelTimer, **KeInitializeTimerEx**, **KeReadStateTimer**, **KeSetTimer**, **KeSetTimerEx**, **KeWaitForMultipleObjects**, **KeWaitForSingleObject**

KeInitializeTimerEx

```
VOID
  KeInitializeTimerEx(
  IN PKTIMER  Timer,
  IN TIMER_TYPE  Type
  );
```

KeInitializeTimerEx initializes an extended kernel timer object.

Parameters

Timer

Pointer to a timer object, for which the caller provides the storage.

Type

Specifies the type of the timer object, either **NotificationTimer** or **SynchronizationTimer**.

Include

wdm.h or *ntddk.h*

Comments

The timer object is initialized with a not signaled state.

Storage for a timer object must be resident: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in non-paged pool allocated by the caller.

When a notification timer expires, all waiting threads are released and the timer remains in the signaled state until it is explicitly reset. When a synchronization timer expires, it is set to a signaled state until a single waiting thread is released and then the timer is reset to a not signaled state.

Callers of **KeInitializeTimerEx** should be running at IRQL DISPATCH_LEVEL or lower. It is best to initialize timers at IRQL PASSIVE_LEVEL.

Use **KeSetTimer** or **KeSetTimerEx** to define when the timer will expire.

See Also

KeCancelTimer, KeReadStateTimer, KeSetTimer, KeSetTimerEx, KeWaitForMultipleObjects, KeWaitForSingleObject

KeInsertByKeyDeviceQueue

```
BOOLEAN
  KeInsertByKeyDeviceQueue(
  IN PKDEVICE_QUEUE  DeviceQueue,
  IN PKDEVICE_QUEUE_ENTRY  DeviceQueueEntry,
  IN ULONG  SortKey
  );
```

KeInsertByKeyDeviceQueue acquires the spin lock for the given *DeviceQueue* and queues an entry according to the given sort-key value if the device queue is set to a busy state.

Parameters

DeviceQueue

Pointer to a control object of the device queue type for which the caller provides the storage.

DeviceQueueEntry

Pointer to the device queue entry to be inserted into the device queue according to the specific key value.

SortKey

Specifies the sort-key value that determines the position in the device queue in which to insert the entry.

Include

wdm.h or *ntddk.h*

Return Value

If the device queue is empty, FALSE is returned, meaning the *DeviceQueueEntry* is not inserted in the device queue.

Comments

The given device queue spin lock is acquired and the state of the device queue is checked. If the device queue is set to a busy state, the IRP specified by the *DeviceQueueEntry* is inserted into the device queue according to its sort key value and the device queue spin lock is released.

The new entry is positioned in the device queue after any entries in the queue with sort key values less than or equal to its sort key value and preceding any entries with sort key values that are greater.

If **KeInsertByKeyDeviceQueue** returns FALSE, the caller must begin processing the IRP. A call to **KeInsertDeviceQueue** or **KeInsertByKeyDeviceQueue** when the queue is empty causes the device queue to transition from a not busy state to a busy state.

This routine is for code that queues an I/O request to a device driver.

Callers of **KeInsertByKeyDeviceQueue** must be running at IRQL DISPATCH_LEVEL.

See Also

KeInitializeDeviceQueue, **KeInsertDeviceQueue**, **KeRemoveDeviceQueue**, **KeRemoveEntryDeviceQueue**

KeInsertDeviceQueue

```
BOOLEAN
  KeInsertDeviceQueue(
  IN PKDEVICE_QUEUE  DeviceQueue,
  IN PKDEVICE_QUEUE_ENTRY  DeviceQueueEntry
  );
```

KeInsertDeviceQueue acquires the spin lock for the given device queue object and, if the device queue is set to a busy state, queues the given entry.

Parameters

DeviceQueue

Pointer to a control object of type device queue for which the caller provides the storage.

DeviceQueueEntry

Pointer to the device queue entry that is to be inserted.

Include

wdm.h or *ntddk.h*

Return Value

If the device queue is empty, FALSE is returned and the *DeviceQueueEntry* is not inserted in the device queue.

Comments

If the device queue is set to a busy state, the specified *DeviceQueueEntry* is inserted at the tail of the device queue and the device queue spin lock is released.

If **KeInsertDeviceQueue** returns FALSE, the entry was not queued and the caller must begin processing the IRP. A call to **KeInsertDeviceQueue** or **KeInsertByKeyDeviceQueue** when the queue is empty causes the device queue to change from a not busy state to a busy state.

This routine is for code that queues an I/O request to a device driver.

Callers of **KeInsertDeviceQueue** must be running at IRQL DISPATCH_LEVEL.

See Also

KeInitializeDeviceQueue, **KeInsertByKeyDeviceQueue**, **KeRemoveDeviceQueue**, **KeRemoveEntryDeviceQueue**

KeInsertQueueDpc

```
BOOLEAN
  KeInsertQueueDpc(
  IN PRKDPC  Dpc,
  IN PVOID  SystemArgument1,
  IN PVOID  SystemArgument2
  );
```

KeInsertQueueDpc queues a DPC for execution when the IRQL of a processor drops below DISPATCH_LEVEL.

Parameters

Dpc

Pointer to an initialized control object of type DPC for which the caller provides the storage.

SystemArgument1, SystemArgument2

Pointer to a set of two parameters that contain untyped data.

Include

wdm.h or *ntddk.h*

Return Value

If the specified DPC object is not currently in the queue, **KeInsertQueueDpc** queues the DPC and returns TRUE.

Comments

If the specified DPC object is already in the DPC queue, no operation is performed except to return FALSE. Otherwise, the DPC object is inserted in the DPC queue and a software interrupt is requested at IRQL DISPATCH_LEVEL on the current processor.

Note that a given DPC object and the function it represents can each be queued for execution only once at any given moment. If it can make overlapped calls to **KeInsertQueueDpc**, particularly in SMP machines, the caller should protect its DPC object with a spinlock.

The deferred procedure is run when IRQL on the current processor drops below DISPATCH_LEVEL.

Callers of **KeInsertQueueDpc** must be running at IRQL >= DISPATCH_LEVEL.

See Also

KeInitializeDpc, **KeRemoveQueueDpc**

KeLeaveCriticalRegion

```
VOID
  KeLeaveCriticalRegion(
  );
```

KeLeaveCriticalRegion re-enables the delivery of normal kernel-mode APCs that were disabled by a preceding call to **KeEnterCriticalRegion**.

Include

ntddk.h

Comments

Highest-level drivers can call this routine while running in the context of the thread that requested the current I/O operation.

Callers of **KeLeaveCriticalRegion** must be running at IRQL PASSIVE_LEVEL.

See Also

KeEnterCriticalRegion

KeLowerIrql

```
VOID
  KeLowerIrql(
  IN KIRQL  NewIrql
  );
```

KeLowerIrql restores the IRQL on the current processor to its original value.

Parameters

NewIrql

Specifies the IRQL that was returned from **KeRaiseIrql**.

Include

wdm.h or *ntddk.h*

Comments

It is a fatal error to call **KeLowerIrql** using an input *NewIrql* that was not returned by the immediately preceding call to **KeRaiseIrql**.

Callers of **KeLowerIrql** can be running at any IRQL that was passed to **KeRaiseIrql**.

See Also

KeGetCurrentIrql, **KeRaiseIrql**

KePulseEvent

```
NTSTATUS
  KePulseEvent(
  IN PRKEVENT  Event,
  IN KPRIORITY  Increment,
  IN BOOLEAN  Wait
  );
```

KePulseEvent atomically sets an event object to a signaled state, attempts to satisfy as many waits as possible, and then resets the event object to a not signaled state. The previous signal state of the event object is returned as the function value.

Parameters

Event

Pointer to a dispatcher object of type event.

Increment

Specifies the priority increment that is to be applied if setting the event causes a wait to be satisfied.

Wait

Specifies a Boolean value that signifies whether the call to **KePulseEvent** will be immediately followed by a call to one of the kernel-mode wait functions.

Include

ntddk.h

Return Value

The previous signal state of the event object.

Comments

Callers of **KePulseEvent** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeInitializeEvent, **KeReadStateEvent**, **KeResetEvent**, **KeSetEvent**

KeQueryInterruptTime

```
ULONGLONG
KeQueryInterruptTime( );
```

KeQueryInterruptTime returns the current value of the system interrupt-time count.

Include

wdm.h or *ntddk.h*

Return Value

KeQueryInterruptTime returns the current interrupt-time count of 100-nanosecond units.

Comments

KeQueryInterruptTime can be used for performance tuning. This routine returns a finer grained measurement than calling **KeQueryTickCount**. A call to **KeQueryInterruptTime** has considerably less overhead than a call to **KeQueryPerformanceCounter**, as well.

Consequently, interrupt time can be used to measure very fine-grained durations while the system is running because operations that set or reset the system time have no effect on the system interrupt time count.

However, power-management state changes do affect the system interrupt time count. Maintenance of the interrupt time count is suspended during system sleep states. When a subsequent wake state transition occurs, the system updates the interrupt time count to compensate for the approximate duration of such a sleep state.

Callers of **KeQueryInterruptTime** can be running at any IRQL.

See Also

KeQueryPerformanceCounter, **KeQueryTickCount**, **KeQueryTimeIncrement**

KeQueryPerformanceCounter

```
LARGE_INTEGER
  KeQueryPerformanceCounter(
  IN PLARGE_INTEGER  PerformanceFrequency  OPTIONAL
  );
```

KeQueryPerformanceCounter provides the finest grained running count available in the system.

Parameters

PerformanceFrequency

Specifies an optional pointer to a variable that is to receive the performance counter frequency.

Include

wdm.h or *ntddk.h*

Return Value

KeQueryPerformanceCounter returns the performance counter value in units of ticks.

Comments

KeQueryPerformanceCounter always returns a 64-bit integer representing the number of ticks. Accumulating the count begins when the system is booted. The count is in ticks; the frequency is reported by *PerformanceFrequency* if this optional parameter is supplied.

The resolution of the timer used to accumulate the current count can be obtained by specifying *PerformanceFrequency*. For example, if the returned *PerformanceFrequency* is 2 million, each tick is 1/2 millionth of a second. Each 1/*x* millionth increment of the count corresponds to one second of elapsed time.

KeQueryPerformanceCounter is intended for time-stamping packets or for computing performance and capacity measurements. It is not intended for measuring elapsed time, for computing stalls or waits, or for iterations.

Use this routine as infrequently as possible. Depending on the platform, **KeQueryPerformanceCounter** can disable system-wide interrupts for a minimal interval. Consequently, calling this routine frequently or repeatedly, as in an iteration, defeats its purpose of returning very fine-grained, running time-stamp information. Calling this routine too frequently can degrade I/O performance for the calling driver and for the system as a whole.

Callers of **KeQueryPerformanceCounter** can be running at any IRQL.

See Also

KeQueryInterruptTime, **KeQuerySystemTime**, **KeQueryTickCount**, **KeQueryTimeIncrement**

KeQueryPriorityThread

```
KPRIORITY
KeQueryPriorityThread(
  IN PRKTHREAD  Thread
  );
```

KeQueryPriorityThread returns the current priority of a given thread.

Parameters

Thread

Pointer to a dispatcher object of type KTHREAD.

Include

wdm.h or *ntddk.h*

Return Value

KeQueryPriorityThread returns the current priority of the given thread.

Comments

The system-defined range of possible return values runs from zero to 32, inclusive, with zero designating the lowest possible thread priority.

Callers of **KeQueryPriorityThread** must be running at IRQL PASSIVE_LEVEL.

See Also

KeGetCurrentThread, **KeSetBasePriorityThread**, **KeSetPriorityThread**, **PsGetCurrentThread**

KeQuerySystemTime

```
VOID
  KeQuerySystemTime(
  OUT PLARGE_INTEGER  CurrentTime
  );
```

KeQuerySystemTime obtains the current system time.

Parameters

CurrentTime

Pointer to the current time on return from **KeQuerySystemTime**.

Include

wdm.h or *ntddk.h*

Comments

System time is a count of 100-nanosecond intervals since January 1, 1601. System time is typically updated approximately every ten milliseconds. This value is computed for the GMT time zone. To adjust this value for the local time zone use **ExSystemTimeToLocalTime**.

Callers of **KeQuerySystemTime** can be running at any IRQL.

See Also

ExSystemTimeToLocalTime, **KeQueryPerformanceCounter**, **KeQueryTickCount**, **KeQueryTimeIncrement**

KeQueryTickCount

```
VOID
  KeQueryTickCount(
  OUT PLARGE_INTEGER  TickCount
  );
```

KeQueryTickCount maintains a count of the interval timer interrupts that have occurred since the system was booted.

Parameters

TickCount

Pointer to the tick count value on return from **KeQueryTickCount**.

Include

wdm.h or *ntddk.h*

Comments

The *TickCount* value increases by one at each interval timer interrupt while the system is running.

The preferred method of determining elapsed time is by using *TickCount* for relative timing and time stamps.

To determine the absolute elapsed time multiply the returned *TickCount* by the **KeQueryTimeIncrement** return value using compiler support for 64-bit integer operations.

Callers of **KeQueryTickCount** can be running at any IRQL.

See Also

KeQueryInterruptTime, **KeQueryPerformanceCounter**, **KeQueryTimeIncrement**

KeQueryTimeIncrement

```
ULONG
  KeQueryTimeIncrement( );
```

KeQueryTimeIncrement returns the number of 100-nanosecond units that are added to the system time each time the interval clock interrupts.

Include

wdm.h or *ntddk.h*

Comments

Callers of **KeQueryTimeIncrement** can be running at any IRQL.

See Also

KeQueryPerformanceCounter, **KeQuerySystemTime**, **KeQueryTickCount**

KeRaiseIrql

```
VOID
  KeRaiseIrql(
  IN KIRQL  NewIrql,
  OUT PKIRQL  OldIrql
  );
```

KeRaiseIrql raises the hardware priority to a given IRQL value, thereby masking off interrupts of equivalent or lower IRQL on the current processor.

Parameters

NewIrql

Specifies the new IRQL to which the hardware priority is to be raised.

OldIrql

Pointer to the storage for the original (unraised) IRQL value to be used in a subsequent call to **KeLowerIrql**.

Include

wdm.h or *ntddk.h*

Comments

If the new IRQL is less than the current IRQL, a bug check occurs. Otherwise, the current IRQL is set to the specified value.

Callers of this routine can be running at any IRQL. Any caller should restore the original IRQL with **KeLowerIrql** as soon as possible.

A call to **KeLowerIrql** is valid if it specifies *NewIrql* <= *CurrentIrql*. A call to **KeRaiseIrql** is valid if the caller specifies *NewIrql* >= *CurrentIrql*.

See Also

KeGetCurrentIrql, **KeLowerIrql**

KeRaiseIrqlToDpcLevel

```
KIRQL
  KeRaiseIrqlToDpcLevel( );
```

KeRaiseIrqlToDpcLevel raises the hardware priority to IRQL DISPATCH_LEVEL, thereby masking off interrupts of equivalent or lower IRQL on the current processor.

ReturnValue

KeRaiseIrqlToDpcLevel returns the IRQL at which the call occurred.

Include

ntddk.h

Comments

Any caller of **KeRaiseIrqlToDpcLevel** should save the returned IRQL value. Every such caller must restore the original IRQL as quickly as possible by passing this returned IRQL in a subsequent call to **KeLowerIrql**.

Callers of **KeRaiseIrqlToDpcLevel** must be running at IRQL <= DISPATCH_LEVEL. Otherwise, a call to this routine causes a bug check.

See Also

KeGetCurrentIrql, **KeLowerIrql**, **KeRaiseIrql**

KeReadStateEvent

```
LONG
  KeReadStateEvent(
  IN PRKEVENT  Event
  );
```

KeReadStateEvent returns the current state, signaled or not signaled, of a given event object.

Parameters

Event

Pointer to an initialized event object for which the caller provides the storage.

Include

ntddk.h

Return Value

If the event object is currently set to a signaled state, a nonzero value is returned. Otherwise, zero is returned.

Comments

It is also possible to read the state of an event from a driver's interrupt service routine at DIRQL, if the following conditions are met: the driver's event object is resident (probably in its device extension), and any other function that accesses the event synchronizes its access with the ISR.

Callers of **KeReadStateEvent** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeClearEvent, **KeInitializeEvent**, **KeResetEvent**, **KeSetEvent**

KeReadStateMutex

```
LONG
  KeReadStateMutex(
  IN PRKMUTEX  Mutex
  );
```

KeReadStateMutex returns the current state, signaled or not signaled, of a given mutex object.

Parameters

Mutex

Pointer to an initialized mutex object for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Return Value

If the return value is one, the state of the mutex object is signaled.

Comments

Callers of **KeReadStateMutex** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExInitializeFastMutex, **KeInitializeMutex**, **KeReleaseMutex**

KeReadStateSemaphore

```
LONG
  KeReadStateSemaphore(
  IN PRKSEMAPHORE  Semaphore
  );
```

KeReadStateSemaphore returns the current state, signaled or not signaled, of a given semaphore object.

Parameters

Semaphore

Pointer to an initialized semaphore object for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Return Value

If the return value is zero, the semaphore object is set to a not signaled state.

Comments

Callers of **KeReadStateSemaphore** can be running at any IRQL.

See Also

KeInitializeSemaphore, **KeReleaseSemaphore**

KeReadStateTimer

```
BOOLEAN
  KeReadStateTimer(
  IN PKTIMER  Timer
  );
```

KeReadStateTimer reads the current state of a given timer object.

Parameters

Timer

Pointer to an initialized timer object, for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Return Value

If the current state of the timer object is signaled, TRUE is returned.

Comments

Callers of **KeReadStateTimer** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeCancelTimer, **KeInitializeTimer**, **KeSetTimer**

KeRegisterBugCheckCallback

```
BOOLEAN
  KeRegisterBugCheckCallback(
  IN PKBUGCHECK_CALLBACK_RECORD  CallbackRecord,
  IN PKBUGCHECK_CALLBACK_ROUTINE  CallbackRoutine,
  IN PVOID  Buffer,
  IN ULONG  Length,
  IN PUCHAR  Component
  );
```

Device drivers can call **KeRegisterBugCheckCallback** to register their bug-check callback routines. If a system bug check occurs, such a callback usually saves device-state information to be written into the system crash dump file.

Parameters

CallbackRecord

Pointer to a callback record, already initialized with **KeInitializeCallbackRecord**, for which the caller provides nonpaged storage.

CallbackRoutine

Specifies the entry point of the caller-supplied routine to be registered, declared as follows:

```
VOID
(*PKBUGCHECK_CALLBACK_ROUTINE) (
    IN PVOID Buffer,
    IN ULONG Length
    );
```

This caller-supplied routine is responsible for writing driver-determined state information at *Buffer* if a bug check occurs.

Buffer

Pointer to a caller-supplied buffer, which must be allocated from nonpaged pool.

Length

Specifies the size in bytes of the caller-allocated buffer.

Component

Pointer to a zero-terminated ANSI string identifying the caller. Usually, this is the name of the device driver, or possibly of its device.

Include

ntddk.h

Return Value

KeRegisterBugCheckCallback returns TRUE if the caller-supplied routine has been successfully added to the set of registered bug-check callbacks.

Comments

KeRegisterBugCheckCallback sets up a driver-supplied routine to be called if a bug check occurs so a device driver can save state information, such as the contents of device registers, that would not otherwise be saved in a system crash-dump file.

A driver-supplied bug-check callback routine writes whatever information the driver designer chooses into the memory at *Buffer*. The format of the data written at *Buffer* is driver-determined. This memory cannot be freed unless the driver first calls **KeDeregisterBugCheckCallback**. Like the driver-allocated memory at *Buffer*, such a bug-check callback routine cannot be pageable.

When the callback routine runs, interrupts are disabled. A callback routine cannot allocate resources, such as memory, because the system is being shut down. A bug-check callback also cannot use synchronization mechanisms, such as a spin lock. However, it should not need to acquire synchronization resources because other driver routines are effectively disabled while the system is being shut down for a bug check. The callback routine can safely call the HAL's READ_PORT_*XXX* and/or READ_REGISTER_*XXX* to collect state information from the device and transfer this data to the driver-allocated buffer. It can call any **Ke***Xxx* or **Hal***Xxx* that neither allocates memory nor acquires a synchronization resource.

The given *Component* string should identify the driver to aid in crash-dump debugging. During driver development, the *Component* identifier can be passed to the debugger to select only that component's dump data for examination. A bug-check callback routine also can be debugged.

Callers of **KeRegisterBugCheckCallback** can be running at any IRQL. Usually, a device driver is running at IRQL PASSIVE_LEVEL in its DriverEntry routine when it calls **KeRegisterBugCheckCallback**.

See Also

ExAllocatePool, KeBugCheck, KeBugCheckEx, KeDeregisterBugCheckCallback, KeInitializeCallbackRecord

KeReleaseMutex

```
LONG
  KeReleaseMutex(
  IN PRKMUTEX  Mutex,
  IN BOOLEAN  Wait
  );
```

KeReleaseMutex releases a given mutex object, specifying whether the caller is to call one of **KeWait***Xxx* as soon as **KeReleaseMutex** returns control.

Parameters

Mutex

Pointer to an initialized mutex object for which the caller provides the storage.

Wait

Specifies whether or not the call to **KeReleaseMutex** is to be immediately followed by a call to one of **KeWait***Xxx*.

Include

wdm.h or *ntddk.h*

Return Value

If the return value is zero, the mutex object was released and attained a state of signaled.

Comments

For better performance, use the **Ex..FastMutex** routines instead of the **Ke..Mutex**. However, a fast mutex cannot be acquired recursively, as a kernel mutex can.

If the mutex object attains a signaled state, an attempt is made to satisfy a wait for the mutex object.

A mutex object can be released only by the thread currently holding the mutex. If an attempt is made to release a mutex not held by the thread, a bug check occurs. An attempt to release a mutex object whose current state is signaled also causes a bug check to occur.

When a mutex object attains a signaled state, it is removed from the list of mutexes held by that thread. If the thread's owned mutex list does not contain any more entries, the thread's original priority is restored.

If the value of the *Wait* parameter is TRUE, the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Therefore, the call to **KeReleaseMutex** must be followed immediately by a call to one of **KeWait***Xxx*.

This allows the caller to release a mutex and wait as one atomic operation, preventing a possibly superfluous context switch. However, a caller cannot wait at raised IRQL nor in an arbitrary thread context for a nonzero interval on a mutex object.

If a mutex is acquired recursively, the holding thread must call **KeReleaseMutex** as many times as it acquired the mutex to set it to the signaled state.

Callers of **KeReleaseMutex** must be running at IRQL PASSIVE_LEVEL.

See Also

ExReleaseFastMutex, **ExReleaseFastMutexUnsafe**, **KeInitializeMutex**, **KeReadStateMutex**, **KeWaitForMultipleObjects**, **KeWaitForMutexObject**, **KeWaitForSingleObject**

KeReleaseSemaphore

```
LONG
  KeReleaseSemaphore(
  IN PRKSEMAPHORE  Semaphore,
  IN KPRIORITY  Increment,
  IN LONG  Adjustment,
  IN BOOLEAN  Wait
  );
```

KeReleaseSemaphore releases a given semaphore object. This routine supplies a run-time priority boost for waiting threads. If this call sets the semaphore to the signaled state, the semaphore count is augmented by the given value. The caller can also specify whether it will call one of the **KeWait***Xxx* routines as soon as **KeReleaseSemaphore** returns control.

Parameters

Semaphore

Pointer to an initialized semaphore object for which the caller provides the storage.

Increment

Specifies the priority increment to be applied if releasing the semaphore causes a wait to be satisfied.

Adjustment

Specifies a value to be added to the current semaphore count. This value must be positive.

Wait

Specifies whether the call to **KeReleaseSemaphore** is to be followed *immediately* by a call to one of the **KeWait***Xxx*.

Include

wdm.h or *ntddk.h*

Return Value

If the return value is zero, the previous state of the semaphore object is not signaled.

Comments

Releasing a semaphore object causes the semaphore count to be augmented by the value of the *Adjustment* parameter. If the resulting value is greater than the limit of the semaphore object, the count is not adjusted and an exception, STATUS_SEMAPHORE_COUNT_EXCEEDED, is raised.

Augmenting the semaphore object count causes the semaphore to attain a signaled state, and an attempt is made to satisfy as many waits as possible on the semaphore object.

If the value of the *Wait* parameter is TRUE, the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Therefore, the call to **KeReleaseSemaphore** *must* be followed *immediately* by a call to one of the **KeWait***Xxx*.

This capability allows the caller to release a semaphore and to wait as one atomic operation, preventing a possibly superfluous context switch. However, the caller cannot wait at raised IRQL nor in an arbitrary thread context for a nonzero interval on a semaphore object.

Callers of **KeReleaseSemaphore** must be running at IRQL <= DISPATCH_LEVEL provided that *Wait* is set to FALSE. Otherwise, the caller must be running at IRQL PASSIVE_LEVEL.

See Also

KeInitializeSemaphore, **KeReadStateSemaphore**, **KeWaitForMultipleObjects**, **KeWaitForSingleObject**

KeReleaseSpinLock

```
VOID
  KeReleaseSpinLock(
  IN PKSPIN_LOCK  SpinLock,
  IN KIRQL  NewIrql
  );
```

KeReleaseSpinLock releases a spin lock and restores the original IRQL at which the caller was running.

Parameters

SpinLock

Pointer to a spin lock for which the caller provides the storage.

NewIrql

Specifies the IRQL value saved from the preceding call to **KeAcquireSpinLock**.

Include

wdm.h or *ntddk.h*

Comments

This call is a reciprocal to **KeAcquireSpinLock**. The input *NewIrql* value must be the *OldIrql* returned by **KeAcquireSpinLock**.

Callers of this routine are running at IRQL DISPATCH_LEVEL. On return from **KeReleaseSpinLock**, IRQL is restored to the *NewIrql* value.

See Also

KeAcquireSpinLock, **KeInitializeSpinLock**

KeReleaseSpinLockFromDpcLevel

```
VOID
  KeReleaseSpinLockFromDpcLevel(
  IN PKSPIN_LOCK  SpinLock
  );
```

KeReleaseSpinLockFromDpcLevel releases an executive spin lock.

Parameters

SpinLock

Pointer to an executive spin lock for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Comments

Drivers call **KeReleaseSpinLockFromDpcLevel** to release a spin lock acquired by calling **KeAcquireSpinLockAtDpcLevel**.

It is an error to call **KeReleaseSpinLockFromDpcLevel** if the given spin lock was acquired by calling **KeAcquireSpinLock** because the caller's original IRQL is not restored, which can cause deadlocks or fatal page faults.

Callers of **KeReleaseSpinLockAtDpcLevel** must be running at IRQL DISPATCH_LEVEL.

See Also

KeAcquireSpinLock, **KeAcquireSpinLockAtDpcLevel**, **KeReleaseSpinLock**

KeRemoveByKeyDeviceQueue

```
PKDEVICE_QUEUE_ENTRY
  KeRemoveByKeyDeviceQueue(
  IN PKDEVICE_QUEUE  DeviceQueue,
  IN ULONG  SortKey
  );
```

KeRemoveByKeyDeviceQueue removes an entry, selected according to a sort key value, from a given device queue.

Parameters

DeviceQueue

Pointer to an initialized device queue object for which the caller provides the storage.

SortKey

Specifies the key to be used when searching the *DeviceQueue*.

Include

wdm.h or *ntddk.h*

Return Value

KeRemoveByKeyDeviceQueue returns the device queue entry that was removed; returns NULL if the queue was empty.

Comments

This routine searches for the first entry in the device queue that has a value greater than or equal to the *SortKey*. After this entry is found, this routine removes the entry from the

device queue and returns it. If no such entry is found, then the first entry in the queue is returned. If the device queue is empty, then the device is set to a not busy state and a NULL pointer is returned.

It is an error to call **KeRemoveByKeyDeviceQueue** when the the device queue object is set to a not busy state.

Callers of **KeRemoveByKeyDeviceQueue** must be running at IRQL DISPATCH_LEVEL.

See Also

KeInitializeDeviceQueue, **KeInsertByKeyDeviceQueue**, **KeInsertDeviceQueue**, **KeRemoveDeviceQueue**, **KeRemoveEntryDeviceQueue**

KeRemoveDeviceQueue

```
PKDEVICE_QUEUE_ENTRY
  KeRemoveDeviceQueue(
  IN PKDEVICE_QUEUE  DeviceQueue
  );
```

KeRemoveDeviceQueue removes an entry from the head of a specified device queue.

Parameters

DeviceQueue

Pointer to an initialized device queue object for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Return Value

If the device queue is empty but is set to a busy state, **KeRemoveDeviceQueue** returns NULL.

Comments

The specified device queue spin lock is acquired and the state of the device queue is checked. If the device queue is set to a busy state and an IRP is queued, this routine dequeues the entry and returns a pointer to the IRP. A call to **KeRemoveDeviceQueue** when the device queue object is set to a busy state but no IRPs are queued causes a state change to not busy. The given device queue's spin lock is released.

It is an error to call **KeRemoveDeviceQueue** when the device queue object is set to a not busy state.

Callers of **KeRemoveDeviceQueue** must be running at IRQL DISPATCH_LEVEL.

See Also

KeInitializeDeviceQueue, **KeInsertByKeyDeviceQueue**, **KeInsertDeviceQueue**, **KeRemoveByKeyDeviceQueue**, **KeRemoveEntryDeviceQueue**

KeRemoveEntryDeviceQueue

```
BOOLEAN
  KeRemoveEntryDeviceQueue(
  IN PKDEVICE_QUEUE  DeviceQueue,
  IN PKDEVICE_QUEUE_ENTRY  DeviceQueueEntry
  );
```

KeRemoveEntryDeviceQueue returns whether the specified entry is in the device queue and removes it, if it was queued, from the device queue.

Parameters

DeviceQueue

Pointer to an initialized device queue object for which the caller provides the storage.

DeviceQueueEntry

Pointer to the entry to be removed from the specified *DeviceQueue*.

Include

wdm.h or *ntddk.h*

Return Value

If the *DeviceQueueEntry* is queued, it is removed and **KeRemoveEntryDeviceQueue** returns TRUE.

Comments

The IRQL is set to DISPATCH_LEVEL and the *DeviceQueue* spin lock is acquired.

If the given *DeviceQueueEntry* is not in the queue, the IRP either is already being processed, or the IRP has been canceled. In this case, **KeRemoveEntryDeviceQueue** simply returns FALSE.

The specified *DeviceQueue* spin lock is released and IRQL is restored to its previous value.

Callers of **KeRemoveEntryDeviceQueue** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeInitializeDeviceQueue, **KeInsertByKeyDeviceQueue**, **KeInsertDeviceQueue**, **KeRemoveByKeyDeviceQueue**, **KeRemoveDeviceQueue**

KeRemoveQueueDpc

```
BOOLEAN
  KeRemoveQueueDpc(
  IN PRKDPC  Dpc
  );
```

KeRemoveQueueDpc removes a given DPC object from the system DPC queue.

Parameters

Dpc

Pointer to an initialized DPC object that was queued by calling **KeInsertQueueDpc**.

Include

wdm.h or *ntddk.h*

Return Value

KeRemoveQueueDpc returns TRUE if the DPC object is in the DPC queue. If the given DPC object is not currently in the DPC queue, no operation is performed and FALSE is returned.

Comments

If the given DPC object is currently queued, it is removed from the queue, canceling a call to the associated DPC routine.

Callers of **KeRemoveQueueDpc** can be running at any IRQL.

See Also

KeInitializeDpc, **KeInsertQueueDpc**

KeResetEvent

```
LONG
  KeResetEvent(
  IN PRKEVENT  Event
  );
```

KeResetEvent resets a specified event object to a not signaled state and returns the previous state of that event object.

Parameters

Event

Pointer to an initialized dispatcher object of type event for which the caller provides the storage.

Include

wdm.h or *ntddk.h*

Return Value

KeResetEvent returns the previous state of the given *Event*, nonzero for a signaled state.

Comments

Event is reset to a not signaled state, meaning that its value is set to zero.

Unless the caller uses the value returned by **KeResetEvent**, setting a given event object to a not signaled state using **KeClearEvent** is faster.

Callers of **KeResetEvent** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeClearEvent, KeInitializeEvent, KeReadStateEvent, KeSetEvent, KeWaitForMultipleObjects, KeWaitForSingleObject

KeRestoreFloatingPointState

```
NTSTATUS
  KeRestoreFloatingPointState(
  IN PKFLOATING_SAVE  FloatSave
  );
```

KeRestoreFloatingPointState restores the nonvolatile floating-point context saved by the preceding call to **KeSaveFloatingPointState**.

Parameters

FloatSave

Specifies the pointer passed in the preceding call to **KeSaveFloatingPointState**.

Include

wdm.h or *ntddk.h*

Return Value

KeRestoreFloatingPointState returns STATUS_SUCCESS.

Comments

KeRestoreFloatingPointState is the reciprocal of **KeSaveFloatingPointState**.

Any routine that calls **KeSaveFloatingPointState** *must* call **KeRestoreFloatingPointState** before that routine returns control, and it must be running at the same IRQL as that from which the preceding call to **KeSaveFloatingPointState** occurred. Failure to meet either of these conditions causes a system bug check.

See Also

KeSaveFloatingPointState

KeSaveFloatingPointState

```
NTSTATUS
  KeSaveFloatingPointState(
  OUT PKFLOATING_SAVE  FloatSave
  );
```

KeSaveFloatingPointState saves the nonvolatile floating-point context so the caller can carry out floating-point operations.

Parameters

FloatSave

Pointer to a caller-allocated resident buffer, which must be at least **sizeof**(KFLOATING_SAVE).

Include

wdm.h or *ntddk.h*

Return Value

KeSaveFloatingPointState returns STATUS_SUCCESS if it saved the current thread's floating-point context and set up a fresh floating point context for the caller. Otherwise, it returns one of the following:

STATUS_ILLEGAL_FLOAT_CONTEXT

The system is configured to use floating point emulation, rather than doing FP operations in the processors.

STATUS_INSUFFICIENT_RESOURCES

KeSaveFloatingPointState could not allocate sufficient memory to save the current thread's floating-point context.

Comments

A successful call to **KeSaveFloatingPointState** allows the caller to carry out floating point operations of its own, but such a caller must restore the previous nonvolatile floating-point context as soon as its FP operations are done. Any routine that calls **KeSaveFloatingPointState** *must* call **KeRestoreFloatingPointState** before that routine returns control.

If the call to **KeSaveFloatingPointState** is successful, the data at *FloatSave* is opaque to the caller, which can release the buffer it allocated only after calling **KeRestoreFloatingPointState**.

For performance reasons, drivers should avoid doing any floating point operations unless absolutely necessary. The overhead of saving and restoring the current thread's nonvolatile floating point state degrades the performance of any driver that does floating-point operations.

Callers of **KeSaveFloatingPointState** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeGetCurrentThread, **KeRestoreFloatingPointState**, **PsCreateSystemThread**

KeSetBasePriorityThread

```
LONG
  KeSetBasePriorityThread(
  IN PRKTHREAD  Thread,
  IN LONG  Increment
  );
```

KeSetBasePriorityThread sets the run-time priority, relative to the current process, for a given thread.

Parameters

Thread

Pointer to a dispatcher object of type KTHREAD.

Increment

Is the value to be added to the base priority of the process for the *Thread*.

Include

ntddk.h

Return Value

KeSetBasePriorityThread returns the previous base priority increment of the given thread. The previous base priority increment is defined as the difference between the specified thread's old base priority and the base priority of the thread's process.

Comments

The new base priority is computed by adding the given *Increment*, which can be a negative value, to the base priority of the specified thread's process. The resultant value is stored as the base priority of the specified thread.

Drivers that set up device-dedicated threads with variable priority attributes can call this routine to set such a thread's priority relative to the system process in which the thread is created.

The new base priority is restricted to the priority class of the given thread's process. Therefore, the base priority is not allowed to cross over from a variable priority class to a real-time priority class or vice versa.

Callers of **KeSetBasePriorityThread** must be running at IRQL PASSIVE_LEVEL.

See Also

KeGetCurrentThread, **KeQueryPriorityThread**, **KeSetPriorityThread**

KeSetEvent

```
LONG
  KeSetEvent(
  IN PRKEVENT  Event,
  IN KPRIORITY  Increment,
  IN BOOLEAN  Wait
  );
```

KeSetEvent sets an event object to a signaled state if the event was not already signaled, and returns the previous state of the event object.

Parameters

Event

Pointer to an initialized event object for which the caller provides the storage.

Increment

Specifies the priority increment to be applied if setting the event causes a wait to be satisfied.

Wait

Specifies whether the call to **KeSetEvent** is to be followed immediately by a call to a **KeWait***Xxx*.

Include

wdm.h or *ntddk.h*

Return Value

If the previous state of the event object was signaled, a nonzero value is returned.

Comments

Calling **KeSetEvent** causes the event to attain a signaled state, and therefore, an attempt is made to satisfy as many waits as possible on the event object.

If the *Wait* parameter is TRUE, the return to the caller is executed without lowering IRQL or releasing the dispatcher database spin lock. Therefore, the call to **KeSetEvent** must be followed immediately by a call to one of the **KeWait***Xxx*.

This allows the caller to set an event and wait as one atomic operation, preventing a possibly superfluous context switch. However, the caller cannot wait at raised IRQL nor in an arbitrary thread context for a nonzero interval on an event object.

If *Wait* is set to FALSE, the caller can be running at IRQL <= DISPATCH_LEVEL. Otherwise, callers of **KeSetEvent** must be running at IRQL PASSIVE_LEVEL and in a nonarbitrary thread context.

See Also

KeClearEvent, **KeInitializeEvent**, **KeReadStateEvent**, **KeResetEvent**, **KeWaitForMultipleObjects**, **KeWaitForSingleObject**

KeSetImportanceDpc

```
VOID
  KeSetImportanceDpc(
  IN PRKDPC  Dpc,
  IN KDPC_IMPORTANCE  Importance
  );
```

KeSetImportanceDpc controls how a particular DPC is queued and, to some degree, how soon the DPC routine is run.

Include

ntddk.h

Parameters

Dpc

Pointer to the caller's DPC object, already initialized with **KeInitializeDpc**.

Importance

Specifies one of the following system-defined values:

LowImportance

Queue the DPC at the end of the target DPC queue, but do not start running queued DPC routines immediately.

MediumImportance

Queue the DPC at the end of the target DPC queue and start running the queued DPC routines immediately if possible. This is the system-assigned default value for DPC objects.

HighImportance

Queue the DPC at the front of the target processor's DPC queue and start running the queued DPC routines immediately if possible.

Comments

KeSetImportanceDpc can override the kernel-determined order in which DPC objects are queued for execution. By default, the kernel queues all DPCs at **MediumImportance**. Usually, each DPC is queued on the current processor from which the call to **KeInsertQueueDpc** occurs or, from a device driver's ISR, the call to **IoRequestDpc** occurs.

In general, an NT device driver should not call **KeSetImportanceDpc** to change the default priority of a driver-created DPC object that represents a CustomDpc routine queued by the driver's ISR with **KeInsertQueueDpc**. **MediumImportance** ensures that such a driver's ISR always returns control before that driver 's corresponding CustomDpc routine runs in

an SMP machine. Otherwise, an NT device driver with such a CustomDpc routine must be capable of handling the following conditions:

- Resetting a device driver's DPC to **LowImportance** requires that driver's CustomDpc routine to be capable of handling all post-interrupt processing for more than one execution of the driver's ISR, effectively for any number of interrupts that might occur between executions of its CustomDpc routine.
- Resetting a device driver's DPC object to **HighImportance** on an SMP platform can cause the driver's ISR and CustomDpc routines to be run simultaneously on different processors, and the driver cannot determine when such concurrent executions will occur. Consequently, such a device driver's ISR and CustomDpc routine must be capable of handling any synchronization problems that might occur due to their concurrent executions.

Callers of **KeSetImportanceDpc** can be running at any IRQL.

See Also

IoRequestDpc, **KeInitializeDpc**, **KeInsertQueueDpc**, **KeSetTargetProcessorDpc**, **KeSynchronizeExecution**

KeSetTargetProcessorDpc

```
VOID
  KeSetTargetProcessorDpc(
  IN PRKDPC  Dpc,
  IN CCHAR  Number
  );
```

KeSetTargetProcessorDpc controls on which processor a particular DPC routine subsequently will be queued.

Parameters

Dpc

Pointer to the caller's DPC object, already initialized with **KeInitializeDpc**.

Number

Specifies the zero-based number of the target processor on which the DPC should be queued and executed.

Include

ntddk.h

Comments

KeSetTargetProcessorDpc can be called on SMP platforms to control the target processor on which the caller's DPC will be queued and, consequently, the target processor on which the caller's DPC routine will execute.

By default, the kernel queues all DPCs on the current processor from which the call to **KeInsertQueueDpc** (or, from NT device drivers, to **IoRequestDpc**) occurs. On a uniprocessor platform, calls to **KeSetTargetProcessorDpc** have no effect.

Callers of **KeSetTargetProcessorDpc** can be running at any IRQL.

See Also

IoRequestDpc, **KeGetCurrentProcessorNumber**, **KeInitializeDpc**, **KeInsertQueueDpc**, **KeSetImportanceDpc**

KeSetPriorityThread

```
KPRIORITY
  KeSetPriorityThread(
  IN PKTHREAD  Thread,
  IN KPRIORITY  Priority
  );
```

KeSetPriorityThread sets the run-time priority of a driver-created thread.

Parameters

Thread

Pointer to the driver-created thread.

Priority

Specifies the priority of the driver-created thread, usually to the real-time priority value, LOW_REALTIME_PRIORITY. The value LOW_PRIORITY is reserved for system use.

Include

wdm.h or *ntddk.h*

Return Value

KeSetPriorityThread returns the old priority of the thread.

Comments

If a call to **KeSetPriorityThread** resets the thread's priority to a lower value, execution of the thread can be rescheduled even if it is currently running or is about to be dispatched for execution.

Callers of **KeSetPriorityThread** must be running at IRQL PASSIVE_LEVEL.

See Also

KeGetCurrentThread, KeQueryPriorityThread, KeSetBasePriorityThread

KeSetTimer

```
BOOLEAN
  KeSetTimer(
  IN PKTIMER  Timer,
  IN LARGE_INTEGER  DueTime,
  IN PKDPC  Dpc  OPTIONAL
  );
```

KeSetTimer sets the absolute or relative interval at which a timer object is to be set to a signaled state and, optionally, supplies a CustomTimerDpc routine to be executed when that interval expires.

Parameters

Timer

Pointer to a timer object that was initialized with **KeInitializeTimer** or **KeInitializeTimerEx**.

DueTime

Specifies the absolute or relative time at which the timer is to expire. If the value of the *DueTime* parameter is negative, the expiration time is relative to the current system time. Otherwise, the expiration time is absolute. The expiration time is expressed in system time units (100-nanosecond intervals). Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes.

Dpc

Pointer to a DPC object that was initialized by **KeInitializeDpc**. This parameter is optional.

Include

wdm.h or *ntddk.h*

Return Value

If the timer object was already in the system timer queue, **KeSetTimer** returns TRUE.

Comments

KeSetTimer:

- Computes the expiration time.
- Sets the timer to a not signaled state.
- Inserts the timer object in the system timer queue.

If the timer object was already in the timer queue, it is implicitly canceled before being set to the new expiration time. A call to **KeSetTimer** before the previously specified *DueTime* has expired cancels both the timer and the call to the *Dpc*, if any, associated with the previous call.

If the *Dpc* parameter is specified, a DPC object is associated with the timer object. When the timer expires, the timer object is removed from the system timer queue and its state is set to signaled. If a DPC object was associated with the timer when it was set, the DPC object is inserted in the system DPC queue to be executed as soon as conditions permit after the timer interval expires.

The expiration of the timer ultimately depends on the granularity of the system clock. The value specified for *DueTime* guarantees that the timer object is set to a signaled state on or after the given *DueTime*. However, **KeSetTimer** cannot override the granularity of the system clock, whatever the value specified for *DueTime*.

Only one instantiation of a given DPC object can be queued at any given moment. To avoid potential race conditions, the DPC passed to **KeSetTimer** should *not* be passed to **KeInsertQueueDpc**.

A caller cannot wait at raised IRQL nor in an arbitrary thread context for a timer to expire by calling **KeWait***Xxx*.

Callers of **KeSetTimer** can specify one expiration time for a timer. To set a recurring timer use **KeSetTimerEx**.

Callers of **KeSetTimer** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeCancelTimer, **KeInitializeDpc**, **KeInitializeTimer**, **KeInitializeTimerEx**, **KeReadStateTimer**, **KeSetTimerEx**, **KeWaitForMultipleObjects**, **KeWaitForSingleObject**

KeSetTimerEx

```
BOOLEAN
  KeSetTimerEx(
  IN PKTIMER  Timer,
  IN LARGE_INTEGER  DueTime,
  IN LONG  Period  OPTIONAL,
  IN PKDPC  Dpc  OPTIONAL
  );
```

KeSetTimerEx sets the absolute or relative interval at which a timer object is to be set to a signaled state, optionally supplies a CustomTimerDpc routine to be executed when that interval expires, and optionally supplies a recurring interval for the timer.

Parameters

Timer

Pointer to a timer object that was initialized with **KeInitializeTimer** or **KeInitialize-TimerEx**.

DueTime

Specifies the absolute or relative time at which the timer is to expire. If the value of the *DueTime* parameter is negative, the expiration time is relative to the current system time. Otherwise, the expiration time is absolute. The expiration time is expressed in system time units (100-nanosecond intervals). Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes.

Period

Specifies an optional period for the timer in milliseconds. Must be less than or equal to MAXLONG.

Dpc

Pointer to a DPC object that was initialized by **KeInitializeDpc**. This parameter is optional.

Include

wdm.h or *ntddk.h*

Return Value

If the timer object was already in the system timer queue, **KeSetTimerEx** returns TRUE.

Comments

KeSetTimerEx:

- Computes the expiration time.
- Sets the timer to a not signaled state.
- Sets the recurring interval for the timer, if one was specified.
- Inserts the timer object in the system timer queue.

If the timer object was already in the timer queue, it is implicitly canceled before being set to the new expiration time. A call to **KeSetTimerEx** before the previously specified *DueTime* has expired cancels both the timer and the call to the *Dpc*, if any, associated with the previous call.

The expiration of the timer ultimately depends on the granularity of the system clock. The value specified for *DueTime* guarantees that the timer object is set to a signaled state on or after the given *DueTime*. However, **KeSetTimerEx** cannot override the granularity of the system clock, whatever the value specified for *DueTime*.

If the *Dpc* parameter is specified, a DPC object is associated with the timer object. When the timer expires, the timer object is removed from the system timer queue and it is set to a signaled state. If a DPC object was associated with the timer when it was set, the DPC object is inserted in the system DPC queue to be executed as soon as conditions permit after the timer interval expires.

A DPC routine cannot deallocate a periodic timer. A DPC routine can deallocate a nonperiodic timer.

Only one instantiation of a given DPC object can be queued at any given moment. To avoid potential race conditions, the DPC passed to **KeSetTimerEx** should *not* be passed to **KeInsertQueueDpc**.

A caller cannot wait at raised IRQL nor in an arbitrary thread context for a timer to expire by calling **KeWait***Xxx*.

Callers of **KeSetTimerEx** must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeCancelTimer, **KeInitializeDpc**, **KeInitializeTimer**, **KeInitializeTimerEx**, **KeReadStateTimer**, **KeWaitForMultipleObjects**, **KeWaitForSingleObject**

KeStallExecutionProcessor

```
VOID
  KeStallExecutionProcessor(
  IN ULONG  MicroSeconds
  );
```

KeStallExecutionProcessor stalls the caller on the current processor for the given interval.

Parameters

MicroSeconds

Specifies the number of microseconds to stall.

Include

wdm.h or *ntddk.h*

Comments

KeStallExecutionProcessor is a processor-dependent routine that busy-waits for at least the specified number of microseconds, but not significantly longer.

This routine is for use by device drivers and other software that must wait for an interval of less than a clock tick but more than for a few instructions. Drivers that call this routine should minimize the number of microseconds they specify (no more than 50). If a driver must wait for a longer interval, it should use another synchronization mechanism.

Callers of **KeStallExecutionProcessor** can be running at any IRQL.

See Also

KeDelayExecutionThread, **KeWaitForMultipleObjects**, **KeWaitForSingleObject**

KeSynchronizeExecution

```
BOOLEAN
  KeSynchronizeExecution(
  IN PKINTERRUPT  Interrupt,
  IN PKSYNCHRONIZE_ROUTINE  SynchronizeRoutine,
  IN PVOID  SynchronizeContext
  );
```

KeSynchronizeExecution synchronizes the execution of a given routine with that of the ISR associated with the given interrupt object pointer.

Parameters

Interrupt

Is a pointer to a set of interrupt objects. This pointer was returned by **IoConnectInterrupt**.

SynchronizeRoutine

Is the entry point for a caller-supplied SynchCritSection routine whose execution is to be synchronized with the execution of the ISR associated with the interrupt objects. A *SynchronizeRoutine* is declared as follows:

```
BOOLEAN
(*PKSYNCHRONIZE_ROUTINE) (
    IN PVOID SynchronizeContext
    );
```

SynchronizeContext

Pointer to a caller-supplied context area to be passed to the *SynchronizeRoutine* when it is called.

Include

wdm.h or *ntddk.h*

Return Value

KeSynchronizeExecution returns TRUE if the operation succeeds.

Comments

When this routine is called, the following occurs:

1. The IRQL is raised to the *SynchronizeIrql* specified in the call to **IoConnectInterrupt**.
2. Access to *SynchronizeContext* is synchronized with the corresponding ISR by acquiring the associated interrupt object spin lock.
3. The specified *SynchronizeRoutine* is called with the input pointer to *SynchronizeContext*.

The caller-supplied *SynchronizeRoutine* runs at DIRQL, so it must execute very quickly.

Callers of **KeSynchronizeExecution** must be running at IRQL <= DIRQL, that is, less than or equal to the value of the *SynchronizeIrql* parameter specified when the caller registered its ISRs with **IoConnectInterrupt**.

See Also

IoConnectInterrupt

KeWaitForMultipleObjects

```
NTSTATUS
  KeWaitForMultipleObjects(
  IN ULONG  Count,
  IN PVOID  Object[],
  IN WAIT_TYPE  WaitType,
  IN KWAIT_REASON  WaitReason,
  IN KPROCESSOR_MODE  WaitMode,
  IN BOOLEAN  Alertable,
  IN PLARGE_INTEGER  Timeout  OPTIONAL,
  IN PKWAIT_BLOCK  WaitBlockArray  OPTIONAL
  );
```

KeWaitForMultipleObjects routine puts the current thread into an alertable or nonalertable wait state until any or all of a number of dispatcher objects are set to a signaled state or (optionally) until the wait times out.

Parameters

Count

Specifies the number of objects to be waited on.

Object

Points to an array of pointers to dispatcher objects (events, mutexes, semaphores, threads, and timers) for which the caller supplies the storage.

WaitType

Specifies either **WaitAll**, indicating that all of the specified objects must attain a signaled state before the wait is satisfied; or **WaitAny**, indicating that any one of the objects must attain a signaled state before the wait is satisfied.

WaitReason

Specifies the reason for the wait. Drivers should set this value to **Executive** or, if the driver is doing work on behalf of a user and is running in the context of a user thread, to **UserRequest**.

WaitMode

Specifies whether the caller waits in **KernelMode** or **UserMode**. Intermediate and lowest level drivers should specify **KernelMode**. The caller must specify **KernelMode** if the set of objects waited on includes a mutex.

Alertable

Specifies a Boolean value that indicates whether the thread can be alerted while it is in the waiting state.

Timeout

Points to an absolute or relative value representing the upper limit for the wait. A negative value specifies an interval relative to the current system time. The value should be expressed in units of 100 nanoseconds. Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes.

WaitBlockArray

Points to an optional array of wait blocks that describe the wait operation.

Include

wdm.h or *ntddk.h*

Return Value

KeWaitForMultipleObjects can return one of the following:

STATUS_SUCCESS

Depending on the specified *WaitType*, one or all of the dispatcher objects in the *Object* array satisfied the wait.

STATUS_ALERTED

The wait is completed because of an alert to the thread.

STATUS_USER_APC

A user APC was delivered to the current thread before the specified *Timeout* interval expired.

STATUS_TIMEOUT

A time-out occurred before the specified set of wait conditions was met. This value can be returned when an explicit time-out value of zero is specified, but the specified set of wait conditions cannot be met immediately.

If **KeWaitForMultipleObjects** returns STATUS_SUCCESS and if **WaitAny** is specified as the *WaitType*, **KeWaitForMultipleObjects** also returns the zero-based index of the object that satisfied the wait at NTSTATUS.

Comments

Each thread object has a built-in array of wait blocks that can be used to wait on several objects concurrently. Whenever possible, the built-in array of wait blocks should be used in a wait-multiple operation because no additional wait block storage needs to be allocated and later deallocated. However, if the number of objects that must be waited on concurrently is greater than the number of built-in wait blocks, use the *WaitBlockArray* parameter to specify an alternate set of wait blocks to be used in the wait operation.

If the *WaitBlockArray* parameter is NULL, the *Count* parameter must be less than or equal to THREAD_WAIT_OBJECTS or a bug check will occur.

If the *WaitBlockArray* pointer is nonNULL, the *Count* parameter must be less than or equal to MAXIMUM_WAIT_OBJECTS or a bug check will occur.

The current state for each of the specified objects is examined to determine whether the wait can be satisfied immediately. If the necessary side effects are performed on the objects, an appropriate value is returned.

If the wait cannot be satisfied immediately and either no time-out value or a nonzero time-out value has been specified, the current thread is put in a waiting state and a new thread is selected for execution on the current processor. If no *Timeout* is supplied, the calling thread will remain in a wait state until the conditions specified by *Object* and *WaitType* are satisfied.

If *Timeout* is specified, the wait will be automatically satisfied if none of the specified wait conditions is met when the given interval expires.

A *Timeout* value of zero allows the testing of a set of wait conditions, conditionally performing any side effects if the wait can be immediately satisfied, as in the acquisition of a mutex.

The *Alertable* parameter specifies whether the thread can be alerted and its wait state consequently aborted. If the value of this parameter is FALSE then the thread cannot be alerted, no matter what the value of the *WaitMode* parameter or the origin of the alert. The only exception to this rule is that of a terminating thread. A thread is automatically made alertable, for instance, when terminated by a user with a CTRL+C.

If the value of *Alertable* is TRUE and one of the following conditions exists, the thread will be alerted:

1. If the origin of the alert is an internal, undocumented kernel-mode routine used to alert threads.
2. If the origin of the alert is a user-mode APC and the value of the *WaitMode* parameter is **UserMode**.

In the first of these two cases, the thread's wait is satisfied with a completion status of STATUS_ALERTED; in the second case, it is satisfied with a completion status of STATUS_USER_APC.

The thread must be alertable for a user-mode APC to be delivered. This is not the case for kernel-mode APCs. A kernel-mode APC can be delivered and executed even though the thread is not alerted. Once the APC's execution completes, the thread's wait will resume. A thread is never alerted nor is its wait aborted by the delivery of a kernel-mode APC.

The delivery of kernel-mode APCs to a waiting thread does not depend on whether the thread can be alerted, but it depends on other conditions. If the kernel-mode APC is a special kernel-mode APC, then the APC is delivered provided that IRQL < APC_LEVEL. If the kernel-mode APC is a normal kernel-mode APC, then the APC is delivered provided that the following three conditions hold:

1. IRQL < APC_LEVEL.
2. No kernel APC is in progress.
3. The thread is not in a critical section.

A special consideration applies when the *Object* parameter passed to **KeWaitForMultipleObjects** is a mutex. If the dispatcher object waited on is a mutex, APC delivery is the same as for all other dispatcher objects *during the wait.* However, once **KeWaitForMultipleObjects** returns with STATUS_SUCCESS and the thread actually holds the mutex, only special kernel-mode APCs are delivered. Delivery of all other APCs, both kernel-mode and user-mode, is disabled. This restriction on the delivery of APCs persists until the mutex is released.

If the *WaitMode* parameter is **UserMode**, the kernel stack can be swapped out during the wait. Consequently, a caller must *never* attempt to pass parameters on the stack when calling **KeWaitForMultipleObjects** with the **UserMode** argument. If you allocate the event on the stack, you must set the *WaitMode* parameter to **KernelMode**.

It is especially important to check the return value of **KeWaitForMultipleObjects** when the *WaitMode* parameter is **UserMode** or *Alertable* is TRUE, because **KeWaitForMultipleObjects** might return early with a status of STATUS_USER_APC or STATUS_ALERTED.

All long term waits that can be aborted by a user should be **UserMode** waits and *Alertable* should be set to FALSE.

Where possible, *Alertable* should be set to FALSE and *WaitMode* should be set to **KernelMode**, in order to reduce driver complexity. The principal exception to this is when the wait is a long term wait.

Callers of **KeWaitForMultipleObjects** can be running at IRQL <= DISPATCH_LEVEL. However, the caller cannot wait at raised IRQL for a nonzero interval nor in an arbitrary thread context on any dispatcher object. Therefore callers usually are running at IRQL PASSIVE_LEVEL. A call while running at IRQL = DISPATCH_LEVEL is valid if and only if the caller specifies a *Timeout* of zero. That is, a driver must not wait for a nonzero interval at IRQL = DISPATCH_LEVEL.

See Also

ExInitializeFastMutex, KeInitializeEvent, KeInitializeMutex, KeInitializeSemaphore, KeInitializeTimer, KeWaitForMutexObject, KeWaitForSingleObject, 3.9.5 Treatment of Alerts and APCs by Threads Waiting on Dispatcher Objects

KeWaitForMutexObject

```
NTSTATUS
  KeWaitForMutexObject(
  IN PRKMUTEX  Mutex,
  IN KWAIT_REASON  WaitReason,
  IN KPROCESSOR_MODE  WaitMode,
  IN BOOLEAN  Alertable,
  IN PLARGE_INTEGER  Timeout  OPTIONAL
  );
```

KeWaitForMutexObject routine puts the current thread into an alertable or nonalertable wait state until the given mutex object is set to a signaled state or (optionally) until the wait times out.

Parameters

Mutex

Pointer to an initialized mutex object for which the caller supplies the storage.

WaitReason

Specifies the reason for the wait, which should be set to **Executive**. If the driver is doing work on behalf of a user and is running in the context of a user thread, this parameter should be set to **UserRequest**.

WaitMode

The caller must specify **KernelMode**.

Alertable

Specifies a Boolean value that indicates whether the wait is alertable.

Timeout

Pointer to a time-out value that specifies the absolute or relative time at which the wait is to be completed (optional). A negative value specifies an interval relative to the current time. The value should be expressed in units of 100 nanoseconds. Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes.

Include

wdm.h or *ntddk.h*

Return Value

KeWaitForMutexObject can return one of the following:

STATUS_SUCCESS

The dispatcher object specified by the *Mutex* parameter satisfied the wait.

STATUS_ALERTED

The wait was completed because of an alert to the thread.

STATUS_USER_APC

A user APC was delivered to the current thread before the specified *Timeout* interval expired.

STATUS_TIMEOUT

A time-out occurred before the mutex was set to a signaled state. This value can be returned when an explicit time-out value of zero is specified and the specified set of wait conditions cannot be immediately met.

Comments

KeWaitForMutexObject is a macro that converts to **KeWaitForSingleObject**, which can be used instead.

For better performance, use the **Ex..FastMutex** routines instead of the **Ke..Mutex**. However, a fast mutex cannot be acquired recursively, as a kernel mutex can.

The current state of the given mutex object is examined to determine whether the wait can be satisfied immediately. If so, the necessary side effects are performed on the mutex. Otherwise, the current thread is put in a waiting state and a new thread is selected for execution on the current processor.

The *Alertable* parameter specifies whether the thread can be alerted and its wait state consequently aborted. If the value of this parameter is FALSE then the thread cannot be alerted, no matter what the value of the *WaitMode* parameter or the origin of the alert. The only exception to this rule is that of a terminating thread. A thread is automatically made alertable, for instance, when terminated by a user with a CTRL+C.

If the value of *Alertable* is TRUE and one of the following conditions is present, the thread will be alerted:

1. If the origin of the alert is an internal, undocumented kernel-mode routine used to alert threads.

2. The origin of the alert is a user-mode APC, and the value of the *WaitMode* parameter is **UserMode**.

In the first of these two cases, the thread's wait is satisfied with a completion status of STATUS_ALERTED; in the second case, it is satisfied with a completion status of STATUS_USER_APC.

The thread must be alertable for a user-mode APC to be delivered. This is not the case for kernel-mode APCs. A kernel-mode APC can be delivered and executed even though the thread is not alerted. Once the APC's execution completes, the thread's wait resumes. A thread is never alerted, nor is its wait aborted, by the delivery of a kernel-mode APC.

The delivery of kernel-mode APCs to a waiting thread does not depend on whether the thread can be alerted. If the kernel-mode APC is a special kernel-mode APC, then the APC is delivered provided that IRQL < APC_LEVEL. If the kernel-mode APC is a normal kernel-mode APC, then the APC is delivered provided that the following three conditions hold:

1. IRQL < APC_LEVEL.
2. No kernel APC is in progress.
3. The thread is not in a critical section.

Since **KeWaitForMutexObject** initiates a wait on a mutex object, a special consideration applies. APC delivery is the same for mutexes as for all other dispatcher objects *during the wait* to acquire the mutex. However, once **KeWaitForMutexObject** returns with STATUS_SUCCESS and the thread actually holds the mutex, only special kernel-mode APCs are delivered. Delivery of all other APCs, both kernel-mode and user-mode, is disabled. This restriction on the delivery of APCs persists until the mutex is released.

If the *WaitMode* parameter is **UserMode**, the kernel stack can be swapped out during the wait. Consequently, a caller must *never* attempt to pass parameters on the stack when calling **KeWaitForMutexObject** using the **UserMode** argument.

It is especially important to check the return value of **KeWaitForMutexObject** when the *WaitMode* parameter is **UserMode** or *Alertable* is TRUE, because **KeWaitForMutex-Object** might return early with a status of STATUS_USER_APC or STATUS_ALERTED.

All long term waits that can be aborted by a user should be **UserMode** waits and *Alertable* should be set to FALSE.

Where possible, *Alertable* should be set to FALSE and *WaitMode* should be set to **Kernel-Mode**, in order to reduce driver complexity. The principal exception to this is when the wait is a long term wait. If no *Timeout* is supplied, the calling thread remains in a wait state until the *Mutex* is signaled.

A *Timeout* of zero allows the testing of a set of wait conditions and for conditionally performing any side effects if the wait can be immediately satisfied, such as acquiring the *Mutex*.

Callers of **KeWaitForMutexObject** must be running at IRQL <= DISPATCH_LEVEL. Usually, the caller must be running at IRQL = PASSIVE_LEVEL and in a nonarbitrary thread context. A call while running at IRQL = DISPATCH_LEVEL is valid if and only if the caller specifies a *Timeout* of zero. That is, a driver must not wait for a nonzero interval at IRQL = DISPATCH_LEVEL.

See Also

ExAcquireFastMutex, ExAcquireFastMutexUnsafe, ExInitializeFastMutex, KeBugCheckEx, KeInitializeMutex, KeReadStateMutex, KeReleaseMutex, 3.9.5 Treatment of Alerts and APCs by Threads Waiting on Dispatcher Objects

KeWaitForSingleObject

```
NTSTATUS
  KeWaitForSingleObject(
  IN PVOID  Object,
  IN KWAIT_REASON  WaitReason,
  IN KPROCESSOR_MODE  WaitMode,
  IN BOOLEAN  Alertable,
  IN PLARGE_INTEGER  Timeout  OPTIONAL
  );
```

KeWaitForSingleObject routine puts the current thread into a wait state until the given dispatcher object is set to a signaled state or (optionally) until the wait times out.

Parameters

Object

Pointer to an initialized dispatcher object (event, mutex, semaphore, thread, or timer) for which the caller supplies the storage.

WaitReason

Specifies the reason for the wait. A driver should set this value to **Executive**, unless it is doing work on behalf of a user and is running in the context of a user thread, in which case it should set this value to **UserRequest**.

WaitMode

Specifies whether the caller waits in **KernelMode** or **UserMode**. Lowest-level and intermediate drivers should specify **KernelMode**. If the given *Object* is a mutex, the caller must specify **KernelMode**.

Alertable

Specifies a Boolean value that is TRUE if the wait is alertable and FALSE otherwise.

Timeout

Pointer to a time-out value that specifies the absolute or relative time at which the wait is to be completed (optional). A negative value specifies an interval relative to the current time. The value should be expressed in units of 100 nanoseconds. Absolute expiration times track any changes in the system time; relative expiration times are not affected by system time changes.

Include

wdm.h or *ntddk.h*

Return Value

KeWaitForSingleObject can return one of the following:

STATUS_SUCCESS

The dispatcher object specified by the *Object* parameter satisfied the wait.

STATUS_ALERTED

The wait was completed because of an alert to the thread.

STATUS_USER_APC

A user APC was delivered to the current thread before the specified *Timeout* interval expired.

STATUS_TIMEOUT

A time-out occurred before the object was set to a signaled state. This value can be returned when the specified set of wait conditions cannot be immediately met and *Timeout* is set to zero.

Comments

The current state of the specified *Object* is examined to determine whether the wait can be satisfied immediately. If so, the necessary side effects are performed on the object. Otherwise, the current thread is put in a waiting state and a new thread is selected for execution on the current processor.

The *Alertable* parameter specifies whether the thread can be alerted and its wait state consequently aborted. If the value of this parameter is FALSE then the thread cannot be alerted, no matter what the value of the *WaitMode* parameter or the origin of the alert. The only exception to this rule is that of a terminating thread. Under certain circumstances a

terminating thread can be alerted while it is in the process of winding down. A thread is automatically made alertable, for instance, when terminated by a user with a CTRL+C.

If the value of *Alertable* is TRUE and one of the following conditions is present, the thread will be alerted:

1. If the origin of the alert is an internal, undocumented kernel-mode routine used to alert threads.
2. The origin of the alert is a user-mode APC, and the value of the *WaitMode* parameter is **UserMode**.

In the first of these two cases, the thread's wait is satisfied with a completion status of STATUS_ALERTED; in the second case, it is satisfied with a completion status of STATUS_USER_APC.

The thread must be alertable for a user-mode APC to be delivered. This is not the case for kernel-mode APCs. A kernel-mode APC can be delivered and executed even though the thread is not alerted. Once the APC's execution completes, the thread's wait resumes. A thread is never alerted, nor is its wait aborted, by the delivery of a kernel-mode APC.

The delivery of kernel-mode APCs to a waiting thread does not depend on whether the thread can be alerted. If the kernel-mode APC is a special kernel-mode APC, then the APC is delivered provided that IRQL < APC_LEVEL. If the kernel-mode APC is a normal kernel-mode APC, then the APC is delivered provided that the following three conditions hold:

1. IRQL < APC_LEVEL.
2. No kernel APC is in progress.
3. The thread is not in a critical section.

A special consideration applies when the *Object* parameter passed to **KeWaitForSingleObject** is a mutex. If the dispatcher object waited on is a mutex, APC delivery is the same as for all other dispatcher objects *during the wait*. However, once **KeWaitForSingleObject** returns with STATUS_SUCCESS and the thread actually holds the mutex, only special kernel-mode APCs are delivered. Delivery of all other APCs, both kernel-mode and user-mode, is disabled. This restriction on the delivery of APCs persists until the mutex is released.

If the *WaitMode* parameter is **UserMode**, the kernel stack can be swapped out during the wait. Consequently, a caller must *never* attempt to pass parameters on the stack when calling **KeWaitForSingleObject** using the **UserMode** argument. If you allocate the event on the stack, you must set the *WaitMode* parameter to **KernelMode**.

It is especially important to check the return value of **KeWaitForSingleObject** when the *WaitMode* parameter is **UserMode** or *Alertable* is TRUE, because **KeWaitForSingleObject** might return early with a status of STATUS_USER_APC or STATUS_ALERTED.

All long term waits that can be aborted by a user should be **UserMode** waits and *Alertable* should be set to FALSE.

Where possible, *Alertable* should be set to FALSE and *WaitMode* should be set to **KernelMode**, in order to reduce driver complexity. The principal exception to this is when the wait is a long term wait.

If no *Timeout* is supplied, the calling thread remains in a wait state until the *Object* is signalled.

A *Timeout* of zero allows the testing of a set of wait conditions and for the conditional performance of any side effects if the wait can be immediately satisfied, as in the acquisition of a mutex.

Callers of **KeWaitForSingleObject** must be running at IRQL <= DISPATCH_LEVEL. Usually, the caller must be running at IRQL = PASSIVE_LEVEL and in a nonarbitrary thread context. A call while running at IRQL = DISPATCH_LEVEL is valid if and only if the caller specifies a *Timeout* of zero. That is, a driver must not wait for a nonzero interval at IRQL = DISPATCH_LEVEL.

See Also

ExInitializeFastMutex, KeBugCheckEx, KeInitializeEvent, KeInitializeMutex, KeInitializeSemaphore, KeInitializeTimer, KeWaitForMultipleObjects, KeWaitForMutexObject, 3.9.5 Treatment of Alerts and APCs by Threads Waiting on Dispatcher Objects

CHAPTER 6

Memory Manager Routines

References for the **Mm***Xxx* routines and macros are in alphabetical order.

For an overview of the functionality of these routines and macros, see Chapter 1, *Summary of Kernel-Mode Support Routines*.

ADDRESS_AND_SIZE_TO_SPAN_PAGES

```
ULONG
  ADDRESS_AND_SIZE_TO_SPAN_PAGES(
  IN PVOID Va,
  IN ULONG Size
  );
```

ADDRESS_AND_SIZE_TO_SPAN_PAGES returns the number of pages spanned by the virtual range defined by a virtual address and the size in bytes of a transfer request.

Parameters

Va

Points to the virtual address that is the base of the range.

Size

Specifies the size in bytes of the transfer request.

Include

wdm.h or *ntddk.h*

Return Value

ADDRESS_AND_SIZE_TO_SPAN_PAGES returns the number of pages spanned by the virtual range starting at *Va*.

Comments

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 page size of the current platform.

Callers of **ADDRESS_AND_SIZE_TO_SPAN_PAGES** can be running at any IRQL, but usually run at IRQL DISPATCH_LEVEL.

See Also

MapTransfer

ARGUMENT_PRESENT

```
BOOLEAN
  ARGUMENT_PRESENT(
  IN (CHAR *) ArgumentPointer
  );
```

ARGUMENT_PRESENT is a macro that takes an argument pointer and returns FALSE if the pointer is NULL, TRUE otherwise.

Parameters

ArgumentPointer

The value of the pointer argument to be tested.

Include

wdm.h or *ntddk.h*

Return Value

ARGUMENT_PRESENT returns FALSE if the value of *ArgumentPointer* is NULL, true otherwise.

Comments

This macro can be called in conditional code to determine whether an optional argument has been passed in a call.

Callers of **ARGUMENT_PRESENT** can be running at any IRQL.

BYTE_OFFSET

```
ULONG
  BYTE_OFFSET(
  IN PVOID Va
  );
```

BYTE_OFFSET takes a virtual address and returns the byte offset of that address within the page.

Parameters

Va

Points to the virtual address.

Include

wdm.h or *ntddk.h*

Return Value

BYTE_OFFSET returns the offset portion of the virtual address.

Comments

Callers of **BYTE_OFFSET** can be running at any IRQL.

BYTES_TO_PAGES

```
ULONG
  BYTES_TO_PAGES(
  IN ULONG Size
  );
```

BYTES_TO_PAGES takes the size in bytes of the transfer request and calculates the number of pages required to contain the bytes.

Parameters

Size

Specifies the size in bytes of the transfer request.

Include

wdm.h or *ntddk.h*

Return Value

BYTES_TO_PAGES returns the number of pages required to contain the specified number of bytes.

Comments

The system-defined constant PAGE_SIZE can be used to determine whether a given length in bytes for a transfer is less than the page size of the current platform.

Callers of **BYTES_TO_PAGES** can be running at any IRQL.

See Also

ADDRESS_AND_SIZE_TO_SPAN_PAGES

COMPUTE_PAGES_SPANNED

```
ULONG
  COMPUTE_PAGES_SPANNED(
  IN PVOID Va,
  IN ULONG Size
  );
```

Use **ADDRESS_AND_SIZE_TO_SPAN_PAGES** instead of this macro.

CONTAINING_RECORD

```
PCHAR
  CONTAINING_RECORD(
  IN PCHAR Address,
  IN TYPE Type,
  IN PCHAR Field
  );
```

CONTAINING_RECORD 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.

Parameters

Address

Points to a field in an instance of a structure of type *Type*.

Type

The name of the type of the structure whose base address is to be returned. For example, type IRP.

Field

The name of the field pointed to by *Address* and which is contained in a structure of type *Type*.

Include

wdm.h or *ntddk.h*

Return Value

Returns the address of the base of the structure containing *Field*.

Comments

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.

Callers of **CONTAINING_RECORD** can be running at any IRQL as long as the structure is resident. If a page fault might occur, callers must be at or below IRQL APC_LEVEL.

See Also

FIELD_OFFSET

FIELD_OFFSET

```
LONG
  FIELD_OFFSET(
  N TYPE Type,
  IN PCHAR Field
  );
```

FIELD_OFFSET returns the byte offset of a named field in a known structure type.

Parameters

Type

The name of a known structure type containing *Field*.

Field

The name of a field in a structure of type *Type*.

Include

wdm.h or *ntddk.h*

Return Value

Returns the byte offset of the caller supplied *Field* in the *Type* structure.

Comments

Used by device driver writers to symbolically determine the offset of a known field in a known structure type.

Callers of **FIELD_OFFSET** can be running at any IRQL as long as the structure is resident. If a page fault could occur, callers must be at or below IRQL APC_LEVEL.

See Also

CONTAINING_RECORD

MmAllocateContiguousMemory

```
PVOID
  MmAllocateContiguousMemory(
  IN ULONG NumberOfBytes,
  IN PHYSICAL_ADDRESS HighestAcceptableAddress
  );
```

MmAllocateContiguousMemory allocates a range of physically contiguous, cache-aligned memory from nonpaged pool.

Parameters

NumberOfBytes

Specifies the size in bytes of the block of contiguous memory to be allocated.

HighestAcceptableAddress

Specifies the highest valid physical address the driver can use. For example, if a device can only reference physical memory in the lower 16MB, this value would be set to 0x00000000FFFFFF.

Include

ntddk.h

Return Value

MmAllocateContiguousMemory returns the base virtual address for the allocated memory. If the request cannot be satisfied, NULL is returned.

Comments

MmAllocateContiguousMemory can be called to allocate a contiguous block of physical memory for a long-term internal buffer, usually from the DriverEntry routine.

A device driver that must use contiguous memory should allocate only what it needs during driver initialization because nonpaged pool is likely to become fragmented as the system runs. Such a driver must deallocate the memory if it is unloaded. Contiguous allocations are aligned on an integral multiple of the processor's data-cache-line size to prevent cache and coherency problems.

Callers of **MmAllocateContiguousMemory** must be running at IRQL = PASSIVE_LEVEL.

See Also

AllocateCommonBuffer, **KeGetDcacheFillSize**, **MmAllocateNonCachedMemory**, **MmFreeContiguousMemory**

MmAllocateContiguousMemorySpecifyCache

```
NTKERNELAPI
PVOID
MmAllocateContiguousMemorySpecifyCache (
  IN SIZE_T NumberOfBytes,
  IN PHYSICAL_ADDRESS LowestAcceptableAddress,
  IN PHYSICAL_ADDRESS HighestAcceptableAddress,
  IN PHYSICAL_ADDRESS BoundaryAddressMultiple OPTIONAL,
  IN MEMORY_CACHING_TYPE CacheType
  );
```

MmAllocateContiguousMemorySpecifyCache allocates a range of physically contiguous, cache-aligned memory from non-paged pool.

Parameters

NumberOfBytes

Specifies the number of bytes to allocate.

LowestAcceptableAddress

Specifies the lowest valid physical address driver can use.

HighestAcceptableAddress

Specifies the highest valid physical address driver can use.

BoundaryAddressMultiple

If nonzero, this specifies the address multiple the allocated buffer must not cross.

CacheType

Specifies a MEMORY_CACHING_TYPE value, which indicates the type of caching allowed for the requested memory. The possible values that drivers can use are as follows.

MmNonCached

The requested memory cannot be cached by the processor.

MmCached

The processor may cache the requested memory.

MmWriteCombined

The requested memory cannot be cached, but can be used as a frame buffer by the video port driver.

Include

ntddk.h

Return Value

MmAllocateContiguousMemorySpecifyCache returns the base virtual address for the allocated memory. If the system is unable to allocate the request buffer, the routine returns NULL.

Comments

Drivers use this routine to allocate memory at initialization. For more information, see ***MmAllocateContiguousMemory***.

MmAllocateNonCachedMemory

```
PVOID
  MmAllocateNonCachedMemory(
  IN ULONG NumberOfBytes
  );
```

MmAllocateNonCachedMemory allocates a virtual address range of noncached and cache-aligned memory.

Parameters

NumberOfBytes

Specifies the size in bytes of the range to be allocated.

Include

ntddk.h

Return Value

If the requested memory cannot be allocated, the return value is NULL. Otherwise, it is the base virtual address of the allocated range.

Comments

MmAllocateNonCachedMemory can be called from a DriverEntry routine to allocate a noncached block of virtual memory for various device-specific buffers.

A device driver that must use noncached memory should allocate only what it needs during driver initialization because nonpaged pool is likely to become fragmented as the system runs. Such a driver must deallocate the memory if it is unloaded. Noncached allocations are aligned on an integral multiple of the processor's data-cache-line size to prevent cache and coherency problems.

Callers of **MmAllocateNonCachedMemory** must be running at IRQL < DISPATCH_LEVEL.

See Also

AllocateCommonBuffer, **KeGetDcacheFillSize**, **MmAllocateContiguousMemory**, **MmFreeNonCachedMemory**

MmAllocatePagesForMdl

```
PMDL
MmAllocatePagesForMdl (
  IN PHYSICAL_ADDRESS LowAddress,
  IN PHYSICAL_ADDRESS HighAddress,
  IN PHYSICAL_ADDRESS SkipBytes,
  IN SIZE_T TotalBytes
  );
```

MmAllocatePagesForMdl allocates zero-filled, nonpaged, physical memory pages to an MDL.

Parameters

LowAddress

Specifies the physical address of the start of the first address range from which the allocated pages can come. If **MmAllocatePagesForMdl** can not allocate the requested number of bytes in the first address range, it iterates through additional address ranges. At each

iteration **MmAllocatePagesForMdl** adds the value of *SkipBytes* to the previous start address to obtain the start of the next address range.

HighAddress

Specifies the physical address of the end of the first address range from which the allocated pages can come.

SkipBytes

Specifies the number of bytes to skip from the start of the previous address range from which the allocated pages can come. *SkipBytes* must be an integer multiple of the page size, in bytes.

TotalBytes

Specifies the total number of bytes to allocate for the MDL.

Include

ntddk.h

Return Value

MmAllocatePagesForMdl returns one of the following:

MDL pointer

The MDL pointer maps a set of physical pages in the specified address range. If the requested number of bytes is not available, the MDL maps as much physical memory as is available.

NULL

There are no physical memory pages in the specified address ranges, or there is not enough virtually-contiguous nonpaged memory for the MDL.

Comments

MmAllocatedPagesForMdl is designed to be used by kernel-mode drivers that achieve substantial performance gains if physical memory for a device is allocated in a specific physical address range. A driver for an AGP graphics card is an example of such a driver.

Depending on how much physical memory is currently available in the requested ranges, **MmAllocatedPagesForMdl** might return an MDL that maps less memory than was requested. The routine returns NULL if no memory was allocated. A client should check the amount of memory that is actually allocated to the MDL.

The caller must use **MmFreePagesFromMdl** to release the memory pages that are described by an MDL that was created by **MmAllocatePagesForMdl**. After calling **MmFreePagesFromMdl**, the caller must also call **ExFreePool** to release the memory allocated for the MDL structure.

MmAllocatePagesForMdl runs at IRQL <= APC_LEVEL.

See Also

MmFreePagesFromMdl, **MmMapLockedPages**, **ExFreePool**

MmBuildMdlForNonPagedPool

```
VOID
  MmBuildMdlForNonPagedPool(
  IN OUT PMDL MemoryDescriptorList
  );
```

MmBuildMdlForNonPagedPool fills in the corresponding physical page array of a given MDL for a buffer in nonpaged system space (pool).

Parameters

MemoryDescriptorList

Points to an MDL that supplies a virtual address, byte offset, and length.

Include

wdm.h or *ntddk.h*

Comments

The physical page portion of the MDL is updated as the pages are locked in memory. The input MDL virtual address must be within the nonpaged portion of system space.

Callers of **MmBuildMdlForNonPagedPool** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ExAllocatePool, **ExAllocatePoolWithTag**, **IoAllocateMdl**, **MmCreateMdl**, **MmInitializeMdl**, **MmIsNonPagedSystemAddressValid**, **MmMapIoSpace**, **MmSizeOfMdl**

MmCreateMdl

```
PMDL
  MmCreateMdl(
  IN PMDL MemoryDescriptorList  OPTIONAL,
  IN PVOID Base,
  IN SIZE_T Length
  );
```

MmCreateMdl is exported to support existing driver binaries. Callers should use **IoAllocateMdl** instead.

MmFreeContiguousMemory

```
VOID
  MmFreeContiguousMemory(
  IN PVOID BaseAddress
  );
```

MmFreeContiguousMemory releases a range of physically contiguous memory that was allocated with **MmAllocateContiguousMemory**.

Parameters

BaseAddress

Points to the virtual address of the memory to be freed.

Include

ntddk.h

Comments

This routine is the reciprocal of **MmAllocateContiguousMemory**, and is usually called only when a driver is being unloaded. The input *BaseAddress* must have been returned by a preceding call to **MmAllocateContiguousMemory**.

A device driver that must use contiguous memory should allocate only what it needs during driver initialization because nonpaged pool is likely to become fragmented as the system runs. Such a driver must deallocate the memory if it is unloaded.

Callers of **MmFreeContiguousMemory** must be running at IRQL = PASSIVE_LEVEL.

See Also

MmAllocateContiguousMemory

MmFreeContiguousMemorySpecifyCache

```
NTKERNELAPI
VOID
MmFreeContiguousMemorySpecifyCache (
  IN PVOID BaseAddress,
  IN SIZE_T NumberOfBytes,
  IN MEMORY_CACHING_TYPE CacheType
  );
```

MmFreeContiguousMemorySpecifyCache frees a buffer allocated by **MmAllocate-ContiguousMemorySpecifyCache**.

Parameters

BaseAddress

Specifies the base address of the buffer to be freed. Must match the address returned by **MmAllocateContiguousMemorySpecifyCache**.

NumberOfBytes

Specifies the size in bytes of the buffer to be freed. Must match the size requested when the buffer was allocated by **MmAllocateContiguousMemorySpecifyCache**.

CacheType

Specifies the cache type of the buffer to be freed. Must match the cache type requested when the buffer was allocated by **MmAllocateContiguousMemorySpecifyCache**.

Include

ntddk.h

MmFreeNonCachedMemory

```
VOID
  MmFreeNonCachedMemory(
  IN PVOID BaseAddress,
  IN SIZE_T NumberOfBytes
  );
```

MmFreeNonCachedMemory releases a range of noncached memory that was allocated with **MmAllocateNonCachedMemory**.

Parameters

BaseAddress

Points to the virtual address of the memory to be freed.

NumberOfBytes

Specifies the size of the range to be freed. The value must match the size passed in a preceding call to **MmAllocateNonCachedMemory**.

Include

ntddk.h

Comments

This routine is the reciprocal of **MmAllocateNonCachedMemory**, and is usually called only when a driver is being unloaded.

A device driver that must use noncached memory should allocate only what it needs during driver initialization because nonpaged pool is likely to become fragmented as the system runs. Such a driver must deallocate the memory if it is unloaded.

Callers of **MmFreeNonCachedMemory** must be running at IRQL < DISPATCH_LEVEL.

See Also

MmAllocateNonCachedMemory

MmFreePagesFromMdl

```
VOID
MmFreePagesFromMdl (
  IN PMDL MemoryDescriptorList
  );
```

MmFreePagesFromMdl frees all the physical pages that are described by an MDL that was created by **MmAllocatePagesForMdl**.

Parameters

MemoryDescriptorList

Pointer to an MDL that was created by **MmAllocatePagesForMdl**.

Include

ntddk.h

Comments

MmFreePagesFromMdl can only be used to free the memory pages that are described by an MDL that was created by **MmAllocatePagesForMdl**.

After calling **MmFreePagesFromMdl**, the caller must also call **ExFreePool** to release the memory that was allocated for the MDL structure.

MmFreePagesFromMdl runs at IRQL <= APC_LEVEL.

See Also

ExFreePool, **MmAllocatePagesForMdl**

MmGetMdlByteCount

```
ULONG
  MmGetMdlByteCount(
  IN PMDL Mdl
  );
```

MmGetMdlByteCount returns the length in bytes of the buffer described by a given MDL.

Parameters

Mdl

Points to an MDL.

Include

wdm.h or *ntddk.h*

Return Value

MmGetMdlByteCount returns the byte count of the buffer described by *Mdl*.

Comments

Callers of **MmGetMdlByteCount** can be running at any IRQL. Usually, callers are running at IRQL <= DISPATCH_LEVEL.

See Also

MmGetMdlByteOffset

MmGetMdlByteOffset

```
ULONG
  MmGetMdlByteOffset(
  IN PMDL Mdl
  );
```

MmGetMdlByteOffset returns the byte offset within a page of the buffer described by a given MDL.

Parameters

Mdl

Points to an MDL.

Include

wdm.h or *ntddk.h*

Return Value

MmGetMdlByteOffset returns the offset in bytes.

Comments

Callers of **MmGetMdlByteOffset** can be running at any IRQL. Usually, callers are running at IRQL <= DISPATCH_LEVEL.

See Also

MmGetMdlByteCount, **MmGetMdlVirtualAddress**

MmGetMdlPfnArray

```
PPFN_NUMBER
MmGetMdlPfnArray (
  IN PMDL Mdl
  );
```

MmGetMdlPfnArray returns a pointer to the beginning of the array of physical page numbers associated with the MDL.

Parameters

Mdl

Points to an MDL.

Include

ntddk.h or *wdm.h*

Return Value

A pointer to the beginning of the array of physical page numbers associated with the MDL.

Comments

Callers of **MmGetMdlPfnArray** can be running at any IRQL.

MmGetMdlVirtualAddress

```
PVOID
  MmGetMdlVirtualAddress(
  IN PMDL Mdl
  );
```

MmGetMdlVirtualAddress returns the base virtual address of a buffer described by an MDL. The returned address, used as an index to a physical address entry in the MDL, can be input to **MapTransfer**.

Parameters

Mdl

Points to an MDL that describes the buffer for which to return the initial virtual address.

Include

wdm.h or *ntddk.h*

Return Value

MmGetMdlVirtualAddress returns the starting virtual address of the MDL.

Comments

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-space memory.

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**.

See Also

MapTransfer, **MmGetMdlByteOffset**, **MmIsAddressValid**, **MmIsNonPagedSystem-AddressValid**

MmGetPhysicalAddress

```
PHYSICAL_ADDRESS
  MmGetPhysicalAddress(
  IN PVOID BaseAddress
  );
```

MmGetPhysicalAddress returns the physical address corresponding to a valid virtual address.

Parameters

BaseAddress

Points to the virtual address for which to return the physical address.

Include

ntddk.h

Return Value

MmGetPhysicalAddress returns the physical address that corresponds to the given virtual address.

Comments

Callers of **MmGetPhysicalAddress** can be running at any IRQL, provided that the *BaseAddress* value is valid.

See Also

MmIsAddressValid, **MmIsNonPagedSystemAddressValid**, **MmMapIoSpace**, **MmProbeAndLockPages**

MmGetSystemAddressForMdl

```
PVOID
  MmGetSystemAddressForMdl(
  IN PMDL Mdl
  );
```

MmGetSystemAddressForMdl returns a nonpaged system-space virtual address for the buffer described by the MDL. It maps the physical pages described by a given MDL into system space, if they are not already mapped to system space. This routine is obsolete in Microsoft® Windows® 2000, and is replaced by **MmGetSystemAddressForMdlSafe**.

Parameters

Mdl

Points to a buffer whose corresponding base virtual address is to be mapped.

Include

wdm.h or *ntddk.h*

Return Value

MmGetSystemAddressForMdl returns the base system-space virtual address that maps the physical pages described by the given MDL.

Comments

Drivers of PIO devices call this routine to translate a virtual address range, described by the MDL at **Irp->MdlAddress**, for a user buffer to a system-space address range.

A caller running at IRQL DISPATCH_LEVEL must supply an MDL that maps nonpageable virtual addresses. The input MDL must describe an already locked-down user-space buffer returned by **MmProbeAndLockPages**, a locked-down buffer returned by **MmBuildMdlForNonPagedPool**, or system-space memory allocated from nonpaged pool, contiguous memory, or noncached memory.

The returned base address has the same offset as the virtual address in the MDL.

Callers of **MmGetSystemAddressForMdl** must be running at IRQL <= DISPATCH_LEVEL.

Windows 2000 issues a bug check if the attempt to map to system space fails. On Windows 98, this routine returns NULL in case of failure.

See Also

MmGetSystemAddressForMdlSafe, **MmBuildMdlForNonPagedPool**, **MmProbeAndLockPages**

MmGetSystemAddressForMdlSafe

```
PVOID
  MmGetSystemAddressForMdlSafe(
  IN PMDL Mdl,
  IN MM_PAGE_PRIORITY Priority
  );
```

MmGetSystemAddressForMdlSafe returns a nonpaged system-space virtual address for the buffer described by the MDL. It maps the physical pages described by a given MDL into system space, if they are not already mapped to system space.

Parameters

Mdl

Points to a buffer whose corresponding base virtual address is to be mapped.

Priority

Specifies an MM_PAGE_PRIORITY value which indicates the importance of success under low available PTE conditions. Possible values include **LowPagePriority**, **NormalPagePriority**, and **HighPagePriority**.

- **LowPagePriority** indicates that the mapping request can fail if system is fairly low on resources. An example of this is a non-critical network connection where the driver can handle the mapping failure.
- **NormalPagePriority** indicates that the mapping request can fail if system is very low on resources. An example of this is a non-critical 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 is the paging file path in a driver.

Include

ntddk.h

Return Value

MmGetSystemAddressForMdlSafe returns the base system-space virtual address that maps the physical pages described by the given MDL. If the pages are not already mapped to system space and the attempt to map them fails, NULL is returned.

Comments

Drivers of PIO devices call this routine to translate a virtual address range, described by the MDL at **Irp->MdlAddress**, for a user buffer to a system-space address range.

A caller running at IRQL DISPATCH_LEVEL must supply an MDL that maps nonpageable virtual addresses. The input MDL must describe an already locked-down user-space buffer returned by **MmProbeAndLockPages**, a locked-down buffer returned by **MmBuildMdlForNonPagedPool**, or system-space memory allocated from nonpaged pool, contiguous memory, or noncached memory.

The returned base address has the same offset as the virtual address in the MDL.

Callers of **MmGetSystemAddressForMdlSafe** must be running at IRQL <= DISPATCH_LEVEL.

See Also

MmGetSystemAddressForMdl, **MmBuildMdlForNonPagedPool**, **MmProbeAndLockPages**

MmInitializeMdl

```
VOID
  MmInitializeMdl(
  IN PMDL MemoryDescriptorList,
  IN PVOID BaseVa,
  IN SIZE_T Length
  );
```

MmInitializeMdl initializes the header of an MDL.

Parameters

MemoryDescriptorList

Points to the MDL to be initialized.

BaseVa

Points to the base virtual address of a buffer.

Length

Specifies the length in bytes of the buffer to be described by the MDL.

Include

wdm.h or *ntddk.h*

Comments

Callers of **MmInitializeMdl** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateMdl, **IoBuildPartialMdl**, **IoFreeMdl**, **MmBuildMdlForNonPagedPool**, **MmCreateMdl**, **MmPrepareMdlForReuse**, **MmSizeOfMdl**

MmIsAddressValid

```
BOOLEAN
  MmIsAddressValid(
  IN PVOID VirtualAddress
  );
```

MmIsAddressValid checks whether a page fault will occur for a read or write operation at a given virtual address.

Parameters

VirtualAddress

Points to the virtual address to check.

Include

ntddk.h

Return Value

If no page fault will occur from reading or writing at the given virtual address, **MmIsAddressValid** returns TRUE.

Comments

Even if **MmIsAddressValid** returns TRUE, accessing the address can cause page faults unless the memory has been locked down or the address is a valid nonpaged pool address.

Callers of **MmIsAddressValid** must be running at IRQL <= DISPATCH_LEVEL.

See Also

MmProbeAndLockPages

MmIsNonPagedSystemAddressValid

```
BOOLEAN
  MmIsNonPagedSystemAddressValid(
  IN PVOID VirtualAddress
  );
```

MmIsNonPagedSystemAddressValid is exported to support existing drivers and is obsolete.

MmIsThisAnNtAsSystem

```
BOOLEAN
  MmIsThisAnNtAsSystem( );
```

MmIsThisAnNtAsSystem checks whether the current platform is running a server version of Windows NT®/Windows® 2000.

Include

ntddk.h

Return Value

If the current platform is a server version of Windows NT/Windows 2000, **MmIsThisAnNtAsSystem** returns TRUE.

Comments

Drivers can use this routine during initialization, along with **MmQuerySystemSize**, for sizing estimates of how many resources to allocate. For example, if **MmIsThisAnNtAsSystem** returns TRUE, the caller can increase the number of threads or the number of initially allocated entries for a lookaside list that it creates in medium and large systems.

Callers of **MmIsThisAnNtAsSystem** must be running at IRQL PASSIVE_LEVEL.

See Also

MmQuerySystemSize, **ExInitializeNPagedLookasideList**, **ExInitializePagedLookasideList**

MmLockPagableCodeSection

```
PVOID
  MmLockPagableCodeSection(
  IN PVOID AddressWithinSection
  );
```

MmLockPagableCodeSection locks a section of driver code, containing a set of driver routines marked with a special compiler directive, into system space.

Parameters

AddressWithinSection

Is a symbolic address, usually the entry point of a driver routine, within the pageable section of driver code.

Include

wdm.h or *ntddk.h*

Return Value

MmLockPagableCodeSection returns a handle for the locked-down section of driver code. The handle must be passed subsequently to **MmLockPagableSectionByHandle** or to **MmUnlockPagableImageSection**.

Comments

This routine and its reciprocal, **MmUnlockPagableImageSection**, support drivers that can do the following:

- Defer loading a subset of driver routines into resident memory until incoming I/O requests for the driver's device(s) make it necessary for these routines to process IRPs.
- Make the same subset of driver routines available for paging out when they have completed the processing of I/O requests and no additional requests for the driver's device(s) are currently expected.

MmLockPagableCodeSection, **MmLockPagableSectionByHandle** and **MmUnlockPagableImageSection** are intended for use by device and intermediate drivers that have the following characteristics:

- The driver has code paths that might not be needed while the system is running, but, if they are needed, the driver's code must be resident because it runs in an arbitrary thread context and/or at raised IRQL.
- The driver can determine exactly when the pageable-routines should be loaded and when they can be paged out again.

For example, the system-supplied fault-tolerant disk driver supports the creation of mirror sets, stripe sets, and volume sets. Yet, a particular machine can be configured only with a mirror set, only with a stripe set, only with a volume set, or with any combination of these three possible options. In these circumstances, the system ftdisk driver reduces the size of its loaded image by marking routines that explicitly support mirror, stripe, and volume sets as belonging to pageable-code sections. During driver initialization, pageable-code section(s) are made resident only if the user has configured the disks to have mirror, stripe, and/or volume sets. If the user repartitions the disks dynamically, the ftdisk driver loads any additional pageable-code sections necessary to support any mirror, stripe, and/or volume sets that the user requests.

As other examples, the system-supplied serial and parallel drivers have DispatchCreate and DispatchClose routines that are called when a particular port is opened for exclusive I/O and when the handle for an opened port is released, respectively. Yet, serial and parallel I/O requests are sporadic, determined by which applications the end user is currently running and which application options the end user is currently exercising. In these circumstances, the system serial and parallel drivers reduce the sizes of their loaded images by marking many routines as belonging to a pageable-code section that the DispatchCreate routine makes resident only when the first port is opened for I/O.

Note that each of the preceding system drivers satisfies both criteria for having pageable sections: the driver has code paths that might not be needed while the system is running and the driver can determine exactly when its pageable section should be loaded and can be paged out again.

Note also that calling **MmLockPagableCodeSection** and **MmUnlockPagableImage-Section** reduce the size of each driver's loaded image at a cost to the driver's and system's performance. Drivers of performance-critical devices, such as keyboard and mouse drivers, and drivers of devices that are constantly in use, such as disk drivers, generally should not have pageable sections: the loss in performance is simply not worth the temporary reduction in driver image size.

Because it is an expensive operation to lock down a section, if a pageable-code section is locked down in more than place by a driver, use **MmLockPagableCodeSection** for the first request. Make subsequent lock requests by calling **MmLockPagableSectionByHandle** passing the handle returned by **MmLockPagableCodeSection**. Locking by handle significantly improves driver performance because the memory manager uses the handle to find the section rather than searching a loaded module list. A locked down section is unlocked by calling **MmUnlockPagableImageSection**.

Each driver routine within a pageable code section must be marked with the following compiler directive, supported by all compilers that are compatible with Windows NT/ Windows 2000:

#pragma alloc_text(PAGE*xxxx*, *DriverRoutine*)

where *xxxx* is an optional four-character, unique identifier for the caller's pageable section and *DriverRoutine* is an entry point to be included within the pageable-code section. For compilers that compatible with Windows NT/Windows 2000, the keyword **PAGE** and the driver-determined suffix, which can be up to four characters, are case-sensitive, that is, **PAGE** must be capitalized.

A single call to **MmLockPagableCodeSection** in, for example, a driver's DispatchCreate routine causes the entire section, containing every driver routine marked with the same **PAGE***xxxx* identifier to be locked in system space.

Certain types of driver routines cannot be made part of any driver's pageable section, including the following:

- Never make an ISR pageable. It is possible for a device driver to receive a spurious interrupt even if its device is not in use, particularly if the interrupt vector could be shared. In general, even if a driver can explicitly disable interrupts on its device, an ISR should not be made pageable.
- Never make a DPC routine pageable if the driver cannot control when the DPC is queued, such as any DpcForIsr or CustomDpc routine that might be queued from an ISR. In general, driver routines that run at raised IRQL and that can be called in an arbitrary thread context or in response to a random external event should not be made pageable.
- Never make the DispatchRead or DispatchWrite routine pageable in any driver that might be part of the system paging I/O path. The driver of a disk that might contain the system page file must have DispatchRead and DispatchWrite routines that are resident while the system is running, as must all drivers layered above such a disk driver.

Note that routines in a pageable section marked with the compiler directive **#pragma alloc_text(PAGExxxx, ...)**differ from routines marked with the compiler directive **#pragma alloc_text(INIT, ...)**. The routines in the **INIT** section are not pageable and are discarded as soon as the driver returns from its DriverEntry or its Reinitialize routine, if it has one.

The Memory Manager maintains an internal lock count on any driver's pageable-section. Calls to **MmLockPagableCodeSection** increment this count and the reciprocal **MmUnlockPagableImageSection** decrements the count. A driver's pageable section is not available to be paged out unless this count is zero.

Callers of **MmLockPagableCodeSection** and **MmLockPagableDataSection** must take care to use the former for code sections and the latter for data sections. If the incorrect form of **MmLockPagableXxxxSection** is used, a fatal error will occur on some platforms.

For more information on creating pageable code sections, see *Pageable Code and Data* in the *Kernel-Mode Drivers Design Guide*.

Callers of **MmLockPagableCodeSection** run at IRQL PASSIVE_LEVEL.

See Also

MmUnlockPagableImageSection, **MmPageEntireDriver**, **MmResetDriverPaging**, **MmLockPagableDataSection**, **MmLockPagableSectionByHandle**

MmLockPagableDataSection

```
PVOID
  MmLockPagableDataSection(
  IN PVOID AddressWithinSection
  );
```

MmLockPagableDataSection locks an entire section of driver data into system space.

Parameters

AddressWithinSection

Is the symbolic address of one item of data within the pageable section.

Include

wdm.h or *ntddk.h*

Return Value

MmLockPagableDataSection returns a handle to the section. This handle must be passed subsequently to **MmLockPagableSectionByHandle** or to **MmUnlockPagableImage-Section**.

Comments

This routine, **MmLockPagableSectionByHandle** and **MmUnlockPagableImageSection** are used by drivers to make data pageable but locked in memory on demand.

Data can be pageable if:

- The data is accessed at <DISPATCH_LEVEL.
- The driver uses the data infrequently and predictably.

Drivers for mixer devices use pageable-data sections. Because the driver uses sufficient data to make creating a pageable-data section worthwhile and the driver knows when the data is needed, such a driver uses **MmLockPagableDataSection**, **MmLockPagable-SectionByHandle** and **MmUnlockPagableImageSection** to bring a data section into system space when needed and make it available to be paged out when not needed.

A single call to **MmLockPagableDataSection** causes the entire section, containing the referenced data, to be locked into system space.

It is an expensive operation to lock down a section. If a pageable-data section is locked down in more than one place by a driver, use **MmLockPagableDataSection** for the first request. Make subsequent lock requests by calling **MmLockPagableSectionByHandle**,

passing the handle returned by **MmLockPagableDataSection**. Locking by handle significantly improves driver performance. A locked down section is unlocked by calling **MmUnlockPagableImageSection**.

The memory manager maintains a reference count on the handle to a section. A pageable-data section is only available to be paged out when the reference count is zero. Every lock request increments the count; every unlock request decrements the count. A driver must unlock a section as many times as it locks a section to insure that such a section will be available to be paged out when the section is not needed. A handle is always valid, no matter what the count. If the count on a handle is zero and a call is made to **MmLockPagableSectionByHandle**, the count is set to one, and if the section has been paged out, it will be paged in.

Callers of **MmLockPagableDataSection** and **MmLockPagableCodeSection** must use the former for data sections and the latter for code sections. If the incorrect form of **MmLockPagableXxxxSection** is used, a fatal error will occur on some platforms.

Data in a pageable-data section is marked by a directive available for compilers that are compatible with Windows NT/Windows 2000. To create a pageable data section, use **#pragma data_seg ("PAGE")**, at the beginning of the data module, and **#pragma data_seg ()** at the end of the module. The keyword **PAGE** is case-sensitive, that is, **PAGE** must be capitalized.

Note that there is also a **#pragma data_seg("INIT")** that is used to make data discardable after system initialization. Except for the use of **INIT** rather than **PAGE**, the syntax is the same. However the result is not; use of the **PAGE** directive makes the data section pageable. When the **INIT** directive is used, the data in the section is discarded as soon as the driver returns from its driver entry routine or its reinitialization routine if the driver has one.

For more information about paging data, see *Pageable Code and Data* in the *Kernel-Mode Drivers Design Guide*.

Callers of **MmLockPagableDataSection** run at IRQL PASSIVE_LEVEL.

See Also

MmLockPagableCodeSection, **MmLockPagableSectionByHandle**, **MmPageEntireDriver**, **MmResetDriverPaging**, **MmUnlockPagableImageSection**

MmLockPagableSectionByHandle

```
VOID
  MmLockPagableSectionByHandle(
  IN PVOID ImageSectionHandle
  );
```

MmLockPagableSectionByHandle takes a handle returned by **MmLockPagableDataSection** or **MmLockPagableCodeSection**. This routine checks to see if the referenced section is resident in the caller's address space and if so, simply increments a reference count on the section. If the section is not resident, **MmLockPagableImage** pages in the section, locks it in system space and sets the reference count to one.

Parameters

ImageSectionHandle

Supplies the handle returned by a call to **MmLockPagableCodeSection** or **MmLockPagableDataSection**.

Include

ntddk.h

Comments

If a pageable section is locked down in more than one place by a driver, use **MmLockPagableXxxxSection** for the first request. Make subsequent lock requests by calling **MmLockPagableSectionByHandle** passing the handle returned by **MmLockPagableXxxxSection**. A locked down section is unlocked by calling **MmUnlockPagableImageSection**.

A handle returned from an **MmLockPagableXxxxSection** is valid until a driver is unloaded.

Locking by handle significantly improves driver performance. When **MmLockPagableCodeSection** or **MmLockPagableDataSection** is called, the memory manager walks the entire loaded module list to find the module containing the specified address. This is an expensive operation. Calling **MmLockPagableImageSectionByHandle** reduces this burden because if the caller supplies a handle to the section, the memory manager no longer has to search.

The memory manager maintains a reference count on the handle to the section. A pageable section is only available to be paged out when the reference count is zero. Every lock request increments the count; every unlock request decrements the count. A driver must take care to unlock a section as many times as it locks a section to insure that such a section will be eligible to be paged out when the section is not needed. Once a handle is obtained, it is always valid, no matter what the count until the driver is unloaded. If the count on a handle is zero and a call is made to **MmLockPagableSectionByHandle**, the count is set to one, and if the section has been paged out, it will be paged in.

A driver cannot call **MmLockPagableSectionByHandle** to lock down user buffers passed in IRPs. Use **MmProbeAndLockPages** instead.

For more information about paging code and data, see *Pageable Code and Data* in the *Kernel-Mode Drivers Design Guide*..

Callers of **MmLockPagableSectionByHandle** runs at IRQL PASSIVE_LEVEL.

See Also

MmLockPagableDataSection, MmLockPagableCodeSection, MmProbeAndLockPages, MmPageEntireDriver, MmResetDriverPaging, MmUnlockPagableImageSection

MmMapIoSpace

```
PVOID
  MmMapIoSpace(
  IN PHYSICAL_ADDRESS PhysicalAddress,
  IN ULONG NumberOfBytes,
  IN MEMORY_CACHING_TYPE CacheEnable
  );
```

MmMapIoSpace maps the given physical address range to nonpaged system space.

Parameters

PhysicalAddress

Specifies the starting physical address of the I/O range to be mapped.

NumberOfBytes

Specifies the number of bytes to be mapped.

CacheEnable

Specifies whether the physical address range can be mapped as cached memory. See *MmAllocateContiguousMemorySpecifyCache* for a description of the possible values.

Include

wdm.h or *ntddk.h*

Return Value

MmMapIoSpace returns the base virtual address that maps the base physical address for the range. If space for mapping the range is insufficient, it returns NULL.

Comments

A driver must call this routine during device start-up if it receives translated resources of type **CmResourceTypeMemory**. **MmMapIoResource** maps the physical address returned in the resource list to a logical address through which the driver can access device registers.

For example, drivers of PIO devices that allocate long-term I/O buffers can call this routine to make such a buffer accessible or to make device memory accessible.

Callers of **MmMapIoSpace** must be running at IRQL = PASSIVE_LEVEL.

See Also

HalAllocateCommonBuffer, **HalTranslateBusAddress**, **MmAllocateContiguous-Memory**, **MmAllocateNonCachedMemory**, **MmMapLockedPages**, **MmUnmapIoSpace**

MmMapLockedPages

```
PVOID
  MmMapLockedPages(
  IN PMDL MemoryDescriptorList,
  IN KPROCESSOR_MODE AccessMode
  );
```

MmMapLockedPages maps physical pages described by a given MDL.

Parameters

MemoryDescriptorList

Points to an MDL that has been updated by **MmProbeAndLockPages**.

AccessMode

Specifies the access mode in which to map the MDL, either **KernelMode** or **UserMode**. Lower-level drivers should use **KernelMode**.

Include

wdm.h or *ntddk.h*

Return Value

For Windows NT 4.0 SP3 and earlier versions, **MmMapLockedPages** returns the base virtual address (a page-aligned address) that maps the locked pages for the range described by the MDL. For versions later than Windows NT 4.0 SP3, **MmMapLockedPages** returns the actual virtual address (page-aligned address with offset).

Comments

The MDL in an IRP at **Irp->MdlAddress** does not necessarily map a buffer for the current thread when that IRP is processed by lower-level drivers. Therefore, the user-mode virtual addresses corresponding to the locked-down physical pages can be invisible or invalid.

Callers of **MmMapLockedPages** must be running at IRQL <= DISPATCH_LEVEL if *AccessMode* is **KernelMode**. Otherwise, the caller must be running at IRQL < DISPATCH_LEVEL.

See Also

MmGetSystemAddressForMdl, **MmGetSystemAddressForMdlSafe**, **MmProbeAndLockPages**, **MmUnmapLockedPages**

MmMapLockedPagesSpecifyCache

```
NTKERNELAPI
PVOID
MmMapLockedPagesSpecifyCache (
  IN PMDL MemoryDescriptorList,
  IN KPROCESSOR_MODE AccessMode,
  IN MEMORY_CACHING_TYPE CacheType,
  IN PVOID BaseAddress,
  IN ULONG BugCheckOnFailure,
  IN MM_PAGE_PRIORITY Priority
  );
```

Parameters

MemoryDescriptorList

Specifies the MDL to be mapped. The caller must already have probed and locked the MDL with **MmProbeAndLockPages**.

AccessMode

Specifies the access mode in which to map the MDL. Either **KernelMode** or **UserMode**.

CacheType

Specifies the type of caching allowed for the MDL. See **MmAllocateContiguousMemorySpecifyCache** for a description of the possible values of *CacheType*.

BaseAddress

If *AccessMode* = **UserMode** then this specifies the starting user address to map the MDL to, or NULL to allow the system to choose the starting address. The system may round down the requested address to fit address boundary requirements, so callers must check the return value.

BugCheckOnFailure

Specifies the behavior of the routine if the MDL cannot be mapped because of low system resources. If TRUE, the system bug checks. If FALSE, the routine returns NULL.

Priority

Indicates the importance of success when PTEs are scarce. See **MmGetSystemAddressForMdlSafe** for a description of the possible values for *Priority*.

Include

ntddk.h

Return Value

MmMapLockedPagesSpecifyCache returns the starting address of the mapped pages. If the pages cannot be mapped and *BugCheckOnFailure* is FALSE, the routine returns NULL.

Comments

If *AccessMode* is **UserMode**, then the caller must be running at IRQL <= APC_LEVEL. If *AccessMode* is **KernelMode**, then the caller must be running at IRQL <= DISPATCH_LEVEL.

MmPageEntireDriver

```
VOID
  MmPageEntireDriver(
  IN PVOID AddressWithinSection
  );
```

MmPageEntireDriver causes all of a driver's code and data to be made pageable, overriding the attributes of the various sections that make up the driver's image.

Parameters

AddressWithinSection

Points to a symbolic address within the driver, for example, DriverEntry.

Include

wdm.h or *ntddk.h*

Comments

Use this routine to force a driver to be completely pageable. The driver must not have interrupts connected on its device before making this call since as a result of this call, the entire

driver will be marked as pageable. If the driver is pageable, such code as the ISR code may be paged out after the driver calls **MmPageEntireDriver**.

MmPageEntireDriver may be called multiple times without intervening calls to **MmResetDriverPaging**. However, subsequent calls to **MmPageEntireDriver** do nothing if the driver's pageable status has not been reset by a call to **MmResetDriverPaging**.

For more information about paging an entire driver, see *Pageable Code and Data* in the *Kernel-Mode Drivers Design Guide*..

Callers of **MmPageEntireDriver** run at IRQL PASSIVE_LEVEL.

See Also

MmResetDriverPaging, **MmLockPagableCodeSection**, **MmLockPagableDataSection**, **MmLockPagableSectionByHandle**, **MmUnlockPagableImageSection**

MmResetDriverPaging

```
VOID
  MmResetDriverPaging(
  IN PVOID AddressWithinSection
  );
```

MmResetDriverPaging resets the pageable status of a driver's sections to that specified when the driver was compiled.

Parameters

AddressWithinSection

Points to a symbolic address in the driver, for example, DriverEntry.

Include

wdm.h or *ntddk.h*

Comments

MmResetDriverPaging causes those routines that would not normally be pageable, to be locked into memory. Hence, image sections such as .text and .data will be locked in memory if this routine is called.

A driver that calls this routine must do so before enabling interrupts on its device.

A call to **MmPageEntireDriver** is not a prerequisite to calling this routine. However, calls to **MmResetDriverPaging** do nothing if the driver's image-section attributes have never been overridden by a call to **MmPageEntireDriver**.

For more information about paging an entire driver, see *Pageable Code and Data* in the *Kernel-Mode Drivers Design Guide*..

Callers of **MmResetDriverPaging** must be running at IRQL PASSIVE_LEVEL.

See Also

MmPageEntireDriver, **MmLockPagableCodeSection**, **MmLockPagableDataSection**, **MmLockPagableSectionByHandle**, **MmUnlockPagableImageSection**

MmPrepareMdlForReuse

```
VOID
  MmPrepareMdlForReuse(
  IN PMDL MDL
  );
```

MmPrepareMdlForReuse reinitializes a caller-allocated MDL.

Parameters

MDL

Points to the MDL that will be reused.

Include

wdm.h or *ntddk.h*

Comments

MmPrepareMdlForReuse is called by drivers that allocate a set of MDLs that they use and reuse repeatedly. After each call to **MmPrepareMdlForReuse**, the given MDL must be reinitialized. Very few drivers call this routine.

Callers of **MmPrepareMdlForReuse** must be running at IRQL <= DISPATCH_LEVEL.

See Also

IoAllocateMdl, **IoBuildPartialMdl**, **IoFreeMdl**, **MmCreateMdl**, **MmInitializeMdl**

MmProbeAndLockPages

```
VOID
  MmProbeAndLockPages(
  IN OUT PMDL MemoryDescriptorList,
  IN KPROCESSOR_MODE AccessMode,
  IN LOCK_OPERATION Operation
  );
```

MmProbeAndLockPages probes specified pages, makes them resident, and locks the physical pages mapped by the virtual address range in memory. The MDL is updated to describe the physical pages.

Parameters

MemoryDescriptorList

Points to an MDL that supplies a virtual address, byte offset, and length. The physical page portion of the MDL is updated when the pages are locked in memory.

AccessMode

Specifies the access mode in which to probe the arguments, either **KernelMode** or **UserMode**.

Operation

Specifies the type of operation for which the caller wants the access rights probed and the pages locked, one of **IoReadAccess**, **IoWriteAccess**, or **IoModifyAccess**.

Include

wdm.h or *ntddk.h*

Comments

The highest-level driver in a chain of layered drivers that use direct I/O calls this routine. Drivers that use buffered I/O never call **MmProbeAndLockPages**.

If this routine fails, an exception is raised. Any driver that calls **MmProbeAndLockPages** must handle such an exception.

A lower-level driver cannot attempt to pass such an exception on to a higher-level driver. It cannot assume anything about a higher-level driver's exception handling capabilities. In particular, the driver cannot call **ExRaiseStatus** to pass on such an exception.

Callers of **MmProbeAndLockPages** must be running at IRQL < DISPATCH_LEVEL for pageable addresses, or at IRQL <= DISPATCH_LEVEL for nonpageable addresses.

See Also

MmUnlockPages

MmQuerySystemSize

```
MM_SYSTEM_SIZE
  MmQuerySystemSize( );
```

MmQuerySystemSize returns an estimate of the amount of memory in the system.

Include

wdm.h or *ntddk.h*

Return Value

MmQuerySystemSize returns one of **MmSmallSystem**, **MmMediumSystem**, or **MmLargeSystem**.

Comments

This routine can be called during driver initialization to determine how much memory it is practical to allocate for an internal buffer.

Callers of **MmQuerySystemSize** must be running at IRQL PASSIVE_LEVEL.

See Also

ExAllocatePool, **ExAllocatePoolWithTag**, **ExInitializeNPagedLookasideList**, **ExInitializePagedLookasideList**, **MmIsThisAnNtAsSystem**

MmSizeOfMdl

```
ULONG
  MmSizeOfMdl(
  IN PVOID Base,
  IN SIZE_T Length
  );
```

MmSizeOfMdl returns the number of bytes to allocate for an MDL describing a given address range.

Parameters

Base

Points to the base virtual address for the range.

Length

Supplies the size, in bytes, of the range.

Include

wdm.h or *ntddk.h*

Return Value

MmSizeOfMdl returns the number of bytes required to contain the MDL.

Comments

The given address range must be locked down if it will be accessed at raised IRQL.

Memory for the MDL itself must be allocated from nonpaged pool if the caller subsequently passes a pointer to the MDL while running at IRQL >= DISPATCH_LEVEL.

Callers of **MmSizeOfMdl** run at any IRQL.

See Also

MmCreateMdl, **MmInitializeMdl**

MmUnlockPages

```
VOID
  MmUnlockPages(
  IN PMDL MemoryDescriptorList
  );
```

MmUnlockPages unlocks physical pages described by a given MDL.

Parameters

MemoryDescriptorList

Points to an MDL.

Include

wdm.h or *ntddk.h*

Comments

The memory described by the specified MDL must have been locked previously by a call to **MmProbeAndLockPages**. As the pages are unlocked, the MDL is updated.

Callers of **MmUnlockPages** must be running at IRQL <= DISPATCH_LEVEL.

See Also

MmProbeAndLockPages

MmUnlockPagableImageSection

```
VOID
  MmUnlockPagableImageSection(
  IN PVOID ImageSectionHandle
  );
```

MmUnlockPagableImageSection releases a section of driver code or driver data, previously locked into system space with **MmLockPagableCodeSection**, **MmLockPagableDataSection** or **MmLockPagableSectionByHandle**, so the section can be paged out again.

Parameters

ImageSectionHandle

Is the handle returned by a call to **MmLockPagableCodeSection** or **MmLockPagableDataSection**.

Include

wdm.h or *ntddk.h*

Comments

The handle for a driver's pageable section must not be released if the driver has any outstanding IRPs in its device queue(s) or internal queue(s). A call to **MmUnlockPagableImageSection** restores the pageability of that entire section when there are no more references to the handle for that section.

The memory manager maintains the reference count on the handle to a section. A pageable section is only available to be paged out when the reference count is zero. Every lock request increments the count; every unlock request decrements the count. A driver must unlock a section as many times as it locks a section to make the section available to be paged out.

A handle is always valid, no matter what the count. If the count on a handle is zero and a call is made to **MmLockPagableSectionByHandle**, the count is set to one, and if the section has been paged out, it will be paged in.

In most cases, **MmUnlockPagableImageSection** is called before a driver's Unload routine. That is, a driver with a pageable section is likely to have its DispatchClose and/or DispatchShutdown routine call **MmUnlockPagableImageSection** before its Unload routine is called. However, care should be taken in unloadable drivers to release any pageable section before the driver itself is unloaded from the system.

For more information on paging code and data, see *Pageable Code and Data* in the *Kernel-Mode Drivers Design Guide*..

Callers of **MmUnlockPagableImageSection** must be running at IRQL PASSIVE_LEVEL.

See Also

MmPageEntireDriver, **MmResetDriverPaging**, **MmLockPagableCodeSection**, **MmLockPagableDataSection**, **MmLockPagableSectionByHandle**

MmUnmapIoSpace

```
VOID
  MmUnmapIoSpace(
  IN PVOID BaseAddress,
  IN SIZE_T NumberOfBytes
  );
```

MmUnmapIoSpace unmaps a specified range of physical addresses previously mapped by **MmMapIoSpace**.

Parameters

BaseAddress

Points to the base virtual address to which the physical pages were mapped.

NumberOfBytes

Specifies the number of bytes that were mapped.

Include

wdm.h or *ntddk.h*

Comments

If a driver calls **MmMapIoSpace** during device start-up, it must call **MmUnmapIoSpace** when it receives a PnP stop-device or remove-device IRP for the same device object.

Callers of **MmUnmapIoSpace** must be running at IRQL = PASSIVE_LEVEL.

See Also

MmMapIoSpace

MmUnmapLockedPages

```
VOID
  MmUnmapLockedPages(
  IN PVOID BaseAddress,
  IN PMDL MemoryDescriptorList
  );
```

MmUnmapLockedPages releases a mapping set up by a preceding call to **MmMapLockedPages**.

Parameters

BaseAddress

Points to the base virtual address to which the physical pages were mapped.

MemoryDescriptorList

Points to an MDL.

Include

wdm.h or *ntddk.h*

Comments

Callers of **MmUnmapLockedPages** must be running at IRQL <= DISPATCH_LEVEL if the pages were mapped to system space. Otherwise, the caller must be running at IRQL < DISPATCH_LEVEL.

See Also

MmMapLockedPages

PAGE_ALIGN

```
PVOID
  PAGE_ALIGN(
  IN PVOID Va
  );
```

PAGE_ALIGN returns a page-aligned virtual address for a given virtual address.

Parameters

Va

Points to the virtual address.

Include

wdm.h or *ntddk.h*

Return Value

PAGE_ALIGN returns a pointer to the page-aligned virtual address.

Comments

Callers of **PAGE_ALIGN** can be running at any IRQL.

ROUND_TO_PAGES

```
ULONG_PTR
  ROUND_TO_PAGES(
  IN ULONG_PTR Size
  );
```

ROUND_TO_PAGES takes a size in bytes and rounds it up to the next full page.

Parameters

Size

Specifies the size in bytes to round up to a page multiple.

Include

wdm.h or *ntddk.h*

Return Value

ROUND_TO_PAGES returns the input size rounded up to a multiple of the page size for the current platform.

Comments

Callers of **ROUND_TO_PAGES** can be running at any IRQL.

CHAPTER 7

Object Manager Routines

References for the **Ob***Xxx* routines are in alphabetical order.

For an overview of the functionality of these routines, see Chapter 1, *Summary of Kernel-Mode Support Routines.*

ObDereferenceObject

```
VOID
  ObDereferenceObject(
  IN PVOID  Object
  );
```

ObDereferenceObject decrements the given object's reference count and performs retention checks.

Parameters

Object

Points to the object's body.

Include

wdm.h or *ntddk.h*

Comments

When the reference count for an object reaches zero, a kernel-mode component can remove the object from the system. However, a driver can remove only those objects that it created, and a driver should never attempt to remove any object that it did not create.

A named object with the permanent attribute, such as a directory object created to hold a driver's symbolic link objects, can be deleted as follows:

1. Call **ObDereferenceObject**.
2. Call the appropriate **ZwOpen***Xxx* or **ZwCreate***Xxx* to get a handle for the object, if necessary.
3. Call **ZwMakeTemporaryObject** with the handle.
4. Call **ZwClose** with the handle.

Callers of **ObDereferenceObject** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ZwClose, ZwMakeTemporaryObject

ObGetObjectSecurity

```
NTSTATUS
  ObGetObjectSecurity(
  IN PVOID  Object,
  OUT PSECURITY_DESCRIPTOR  *SecurityDescriptor,
  OUT PBOOLEAN  MemoryAllocated
  );
```

ObGetObjectSecurity gets the security descriptor for a given object.

Parameters

Object

Points to the object.

SecurityDescriptor

Points to a caller-supplied variable that this routine sets to the address of a buffer containing the security descriptor for the given object. If the given object has no security descriptor, this variable is set to NULL on return from **ObGetObjectSecurity**.

MemoryAllocated

Points to a caller-supplied variable that this routine sets to TRUE if it allocated a buffer to contain the security descriptor.

Include

ntddk.h

Return Value

ObGetObjectSecurity either returns STATUS_SUCCESS or an error status, such as STATUS_INSUFFICIENT_RESOURCES if it could not allocate enough memory to return the requested information.

Comments

A successful call to **ObGetObjectSecurity** either returns a self-relative security descriptor in the buffer at **SecurityDescriptor* or it returns NULL at **SecurityDescriptor* if the given object has no security descriptor. For example, any unnamed object, such as an event object, has no security descriptor.

If **ObGetObjectSecurity** returns STATUS_SUCCESS, the caller must save the value returned at *MemoryAllocated*. Such a caller must pass *MemoryAllocated* in a reciprocal call to **ObReleaseObjectSecurity** eventually, thereby restoring the reference count on the security descriptor to its original value and releasing the buffer, if any, that was allocated by **ObGetObjectSecurity**.

Callers of **ObGetObjectSecurity** must be running at IRQL < DISPATCH_LEVEL.

See Also

ObReferenceObjectByHandle, **ObReleaseObjectSecurity**

ObReferenceObject

```
VOID
  ObReferenceObject(
  IN PVOID  Object
  );
```

ObReferenceObject increments the reference count to the given object.

Parameter

Object

Points to the object. The caller obtained this parameter either when it created the object or from a preceding call to **ObReferenceObjectByHandle** after it opened the object.

Include

wdm.h or *ntddk.h*

Comments

ObReferenceObject simply increments the pointer reference count for an object, without making any access checks on the given object, as **ObReferenceObjectByHandle** and **ObReferenceObjectByPointer** do.

ObReferenceObject prevents deletion of the object at least until the driver subsequently calls its reciprocal, **ObDereferenceObject**, or closes the given object. The caller must decrement the reference count with **ObDereferenceObject** as soon as it is done with the object.

When the reference count for an object reaches zero, a kernel-mode component can remove the object from the system. However, a driver can remove only those objects that it created, and a driver should never attempt to remove any object that it did not create.

Callers of **ObReferenceObject** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ObReferenceObjectByHandle, **ObReferenceObjectByPointer**, **ObDereferenceObject**, **ZwClose**

ObReferenceObjectByHandle

```
NTSTATUS
  ObReferenceObjectByHandle(
  IN HANDLE  Handle,
  IN ACCESS_MASK  DesiredAccess,
  IN POBJECT_TYPE  ObjectType  OPTIONAL,
  IN KPROCESSOR_MODE  AccessMode,
  OUT PVOID  *Object,
  OUT POBJECT_HANDLE_INFORMATION  HandleInformation  OPTIONAL
  );
```

ObReferenceObjectByHandle provides access validation on the object handle, and, if access can be granted, returns the corresponding pointer to the object's body.

Parameters

Handle

Specifies an open handle for an object.

DesiredAccess

Specifies the requested types of access to the object. The interpretation of this field is dependent on the object type.

ObjectType

Points to the object type, which can be either of **IoFileObjectType** or **ExEventObjectType**. This parameter can be NULL if *AccessMode* is **KernelMode**.

AccessMode

Specifies the access mode to use for the access check. It must be either **UserMode** or **KernelMode**. Lower-level drivers should specify **KernelMode**.

Object

Points to a variable that receives a pointer to the object's body.

HandleInformation

Points to a structure that receives the handle attributes and the granted access rights for the object.

Include

wdm.h or *ntddk.h*

Return Value

ObReferenceObjectByHandle can return one of the following:

STATUS_SUCCESS
STATUS_OBJECT_TYPE_MISMATCH
STATUS_ACCESS_DENIED
STATUS_INVALID_HANDLE

Comments

A pointer to the object body is retrieved from the object table entry and returned to the caller by means of the *Object* parameter.

If the *AccessMode* parameter is **KernelMode**, the requested access is always allowed. If *AccessMode* is **UserMode**, the requested access is compared to the granted access for the object. Only highest-level drivers can safely specify **UserMode** for the input *AccessMode*.

If the call succeeds, a pointer to the object body is returned to the caller and the pointer reference count is incremented. Incrementing this count prevents the object from being deleted while the pointer is being referenced. The caller must decrement the reference count with **ObDereferenceObject** as soon as it is done with the object.

Callers of **ObReferenceObjectByHandle** must be running at IRQL PASSIVE_LEVEL.

See Also

ObDereferenceObject, **ObReferenceObject**, **ObReferenceObjectByPointer**

ObReferenceObjectByPointer

```
NTSTATUS
  ObReferenceObjectByPointer(
  IN PVOID  Object,
  IN ACCESS_MASK  DesiredAccess,
  IN POBJECT_TYPE  ObjectType,
  IN KPROCESSOR_MODE  AccessMode
  );
```

ObReferenceObjectByPointer increments the pointer reference count for a given object.

Parameters

Object

Points to the object's body.

DesiredAccess

Specifies a mask representing the requested access to the object.

ObjectType

Points to the object type, which can be either of **IoFileObjectType** or **ExEventObjectType**. This parameter can be NULL if *AccessMode* is **KernelMode**.

AccessMode

Indicates the access mode to use for the access check. It must be either **UserMode** or **KernelMode**. Lower-level drivers should specify **KernelMode**.

Include

wdm.h or *ntddk.h*

Return Value

ObReferenceObjectByPointer can return one of the following:

STATUS_SUCCESS
STATUS_OBJECT_TYPE_MISMATCH

Comments

Calling this routine prevents the object from being deleted, possibly by another component's call to **ObDereferenceObject** or **ZwClose**. The caller must decrement the reference count with **ObDereferenceObject** as soon as it is done with the object.

Callers of **ObReferenceObjectByPointer** must be running at IRQL <= DISPATCH_LEVEL.

See Also

ObDereferenceObject, **ObReferenceObject**, **ObReferenceObjectByHandle**, **ZwClose**

ObReleaseObjectSecurity

```
VOID
  ObReleaseObjectSecurity(
  IN PSECURITY_DESCRIPTOR  SecurityDescriptor,
  IN BOOLEAN  MemoryAllocated
  );
```

ObReleaseObjectSecurity is the reciprocal to **ObGetObjectSecurity**.

Parameters

SecurityDescriptor

Points to the buffered security descriptor to be released. The caller obtained this parameter from **ObGetObjectSecurity**

MemoryAllocated

Specifies the value also obtained from **ObGetObjectSecurity**.

Include

ntddk.h

Comments

After a successful call to **ObGetObjectSecurity**, a driver must call **ObReleaseObject-Security** eventually.

ObReleaseObjectSecurity releases any resources that were allocated by **ObGetObject-Security**. It also decrements the reference count on the given security descriptor.

Callers of **ObReleaseObjectSecurity** must be running at IRQL < DISPATCH_LEVEL.

See Also

ObGetObjectSecurity

CHAPTER 8

Process Structure Routines

References for the **Ps***Xxx* routines are in alphabetical order.

For an overview of the functionality of these routines, see Chapter 1, *Summary of Kernel-Mode Support Routines.*

PsCreateSystemThread

```
NTSTATUS
  PsCreateSystemThread(
  OUT PHANDLE  ThreadHandle,
  IN ULONG  DesiredAccess,
  IN POBJECT_ATTRIBUTES  ObjectAttributes  OPTIONAL,
  IN HANDLE  ProcessHandle  OPTIONAL,
  OUT PCLIENT_ID  ClientId  OPTIONAL,
  IN PKSTART_ROUTINE  StartRoutine,
  IN PVOID  StartContext
  );
```

PsCreateSystemThread creates a system thread that executes in kernel mode and returns a handle for the thread.

Parameters

ThreadHandle

Points to a variable that will receive the handle.

DesiredAccess

Specifies the requested types of access to the created thread. This value can be THREAD_ALL_ACCESS or (ACCESS_MASK) 0L for a driver-created thread.

ObjectAttributes

Points to a structure that specifies the object's attributes. OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPENIF, and OBJ_OPENLINK are not valid attributes for a thread object. This value should be NULL for a driver-created thread.

ProcessHandle

Specifies an open handle for the process in whose address space the thread is to be run. The caller's thread must have PROCESS_CREATE_THREAD access to this process. If this parameter is not supplied, the thread will be created in the initial system process. This value should be NULL for a driver-created thread.

ClientId

Points to a structure that receives the client identifier of the new thread. This value should be NULL for a driver-created thread.

StartRoutine

Is the entry point for a driver thread.

StartContext

Supplies a single argument passed to the thread when it begins execution.

Include

wdm.h or *ntddk.h*

Return Value

PsCreateSystemThread returns STATUS_SUCCESS if the thread was created.

Comments

Drivers that create device-dedicated threads call this routine, either when they initialize or when I/O requests begin to come in to such a driver's Dispatch routines. For example, a driver might create such a thread when it receives an asynchronous device control request.

PsCreateSystemThread creates a kernel-mode thread that begins a separate thread of execution within the system. Such a system thread has no TEB or user-mode context and runs only in kernel mode.

If the input *ProcessHandle* is NULL, the created thread is associated with the system process. Such a thread continues running until either the system is shut down or the thread terminates itself by calling **PsTerminateSystemThread**.

Driver routines that run in a process context other than that of the system process should set the OBJ_KERNEL_HANDLE flag within the *Attributes* parameter of **PsCreateSystemThread** before calling it. This restricts the use of the handle returned by **PsCreateSystem-**

Thread to processes running in kernel mode and thereby prevents an unintended access of this handle by the process in whose context the driver is running.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

KeSetBasePriorityThread, **KeSetPriorityThread**, **PsTerminateSystemThread**, **ZwSetInformationThread**

PsGetCurrentProcess

```
PEPROCESS
  PsGetCurrentProcess( );
```

PsGetCurrentProcess returns a pointer to the process of the current thread.

Include

wdm.h or *ntddk.h*

Comments

Highest-level drivers, file systems in particular, can call this routine. Lower-level drivers should call **IoGetCurrentProcess** instead.

PsGetCurrentProcess is the most efficient way to get a pointer to the current process, no matter what environment is being used.

Callers of **PsGetCurrentProcess** must be running at IRQL PASSIVE_LEVEL.

See Also

IoGetCurrentProcess, **PsGetCurrentThread**

PsGetCurrentProcessId

```
HANDLE
  PsGetCurrentProcessId( );
```

PsGetCurrentProcessId identifies the current process.

Include

ntddk.h

Return Value

PsGetCurrentProcessId returns the 4-byte identifier of the current process.

Comments

Highest-level drivers, such as system profilers or IFSs, that register their own callback(s) with **PsSetCreateProcessNotifyRoutine**, **PsSetCreateThreadNotifyRoutine**, and/or **PsSetLoadImageNotifyRoutine** are likely to call **PsGetCurrentProcessId**.

Callers of **PsGetCurrentProcessId** should treat the returned ID as a read-only value.

Callers of **PsGetCurrentProcessId** can be running at any IRQL.

See Also

IoGetCurrentProcess, **PsGetCurrentProcess**, **PsGetCurrentThread**, **PsGetCurrentThreadId**, **PsSetCreateProcessNotifyRoutine**, **PsSetCreateThreadNotifyRoutine**, **PsSetLoadImageNotifyRoutine**

PsGetCurrentThread

```
PETHREAD
  PsGetCurrentThread( );
```

PsGetCurrentThread identifies the current thread.

Include

wdm.h or *ntddk.h*

Return Value

PsGetCurrentThread returns a pointer to the executive thread object representing the currently executing thread.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

KeGetCurrentThread, **PsCreateSystemThread**, **PsGetCurrentProcess**, **PsGetCurrentProcessId**

PsGetCurrentThreadId

```
HANDLE
  PsGetCurrentThreadId( );
```

PsGetCurrentThreadId identifies the current thread.

Include

ntddk.h

Return Value

PsGetCurrentThreadId returns the 4-byte identifier of the current thread.

Comments

Highest-level drivers, such a system profilers or IFSs, that register their own callback with **PsSetCreateThreadNotifyRoutine** are likely to call **PsGetCurrentThreadId**.

Callers of **PsGetCurrentThreadId** should treat the returned ID as a read-only value.

Callers of **PsGetCurrentThreadId** can be running at any IRQL.

See Also

KeGetCurrentThread, **PsGetCurrentProcessId**, **PsGetCurrentThread**, **PsSetCreateThreadNotifyRoutine**

PsGetVersion

```
BOOLEAN
  PsGetVersion(
  PULONG  MajorVersion  OPTIONAL,
  PULONG  MinorVersion  OPTIONAL,
  PULONG  BuildNumber  OPTIONAL,
  PUNICODE_STRING  CSDVersion  OPTIONAL
  );
```

PsGetVersion returns caller-selected information about the current version of Microsoft Windows NT®/Windows® 2000.

Parameters

MajorVersion

Points to a caller-supplied variable that this routine sets to the major version of the operating system. This optional parameter can be NULL.

MinorVersion

Points to a caller-supplied variable that this routine sets to the minor version of the operating system. This optional parameter can be NULL.

BuildNumber

Points to a caller-supplied variable that this routine sets to the current build number of the operating system. This optional parameter can be NULL.

CSDVersion

Points to a caller-allocated buffer in which this routine returns the current service-pack version as a Unicode string only during system driver initialization. This optional parameter can be NULL.

Include

ntddk.h

Return Value

PsGetVersion returns whether the system is a checked or free build, as follows:

Value	Meaning
TRUE (1)	Checked build of the operating system
FALSE (0)	Free build of the operating system

Comments

PsGetVersion returns the requested information, depending on which optional parameter(s) the caller supplies.

To retrieve the current service-pack number, it is easier and more efficient to make an application-level call within the Win32® environment than to call **PsGetVersion** during system driver initialization, which then must parse the string it returns at *CSDVersion*. When the registry is initialized, a driver cannot obtain this string from **PsGetVersion**, but must read the **CmCSDVersionString** value from the registry.

Callers of **PsGetVersion** must be running at IRQL PASSIVE_LEVEL.

PsSetCreateProcessNotifyRoutine

```
NTSTATUS
  PsSetCreateProcessNotifyRoutine(
  IN PCREATE_PROCESS_NOTIFY_ROUTINE  NotifyRoutine,
  IN BOOLEAN  Remove
  );
```

PsSetCreateProcessNotifyRoutine registers a driver-supplied callback that is subsequently notified whenever a process is created or deleted.

Parameters

NotifyRoutine

Specifies the entry point of the caller-supplied process-creation callback.

Remove

Must specify zero.

Include

ntddk.h

Return Value

PsSetCreateProcessNotifyRoutine can return one of the following:

STATUS_SUCCESS

The given *NotifyRoutine* is now registered with the system.

STATUS_INVALID_PARAMETER

The given *NotifyRoutine* has already been registered so this is a redundant call, or the system has reached its limit on registering process-creation callbacks.

Comments

Highest-level drivers can call **PsSetCreateProcessNotifyRoutine** to set up their process-creation notify routines, declared as follows:

```
VOID
(*PCREATE_PROCESS_NOTIFY_ROUTINE) (
    IN HANDLE  ParentId,
    IN HANDLE  ProcessId,
    IN BOOLEAN  Create
    );
```

For example, an IFS or highest-level system-profiling driver might register such a process-creation callback to track the system-wide creation and deletion of processes against the driver's internal state. Windows 2000 registers up to eight such process-creation callbacks. Any driver that successfully registers such a callback *must remain loaded until the system itself is shut down.*

After such a driver-supplied routine is registered, it is called with *Create* set to TRUE just after the initial thread is created within the newly created process designated by the input *ProcessId* handle. The input *ParentId* handle identifies the parent process of the newly created process if it inherits open handles from its parent.

Such a driver's process-notify routine is also called with *Create* set to FALSE, usually when the last thread within each such process has terminated and the process address space is about to be deleted. In very rare circumstances, a driver's process-notify routine can be called only at the destruction of a process in which no thread was ever created.

When it is called, the driver's process-creation notify routine runs at IRQL PASSIVE_LEVEL, either in the context of the initial thread within a newly created process or in the context of a system thread.

Callers of **PsSetCreateProcessNotifyRoutine** must be running at IRQL PASSIVE_LEVEL.

See Also

PsGetCurrentProcessId, PsSetCreateThreadNotifyRoutine, PsSetLoadImageNotifyRoutine

PsSetCreateThreadNotifyRoutine

```
NTSTATUS
  PsSetCreateThreadNotifyRoutine(
  IN PCREATE_THREAD_NOTIFY_ROUTINE  NotifyRoutine
  );
```

PsSetCreateThreadNotifyRoutine registers a driver-supplied callback that is subsequently notified when a new thread is created and when such a thread is deleted.

Parameter

NotifyRoutine

Specifies the entry point of the caller-supplied thread-creation callback.

Include

ntddk.h

Return Value

PsSetCreateThreadNotifyRoutine either returns STATUS_SUCCESS or it returns STATUS_INSUFFICIENT_RESOURCES if it failed the callback registration.

Comments

Highest-level drivers can call **PsSetCreateThreadNotifyRoutine** to set up their thread-creation notify routines, declared as follows:

```
VOID
(*PCREATE_THREAD_NOTIFY_ROUTINE) (
    IN HANDLE  ProcessId,
    IN HANDLE  ThreadId,
    IN BOOLEAN  Create
    );
```

For example, an IFS or highest-level system-profiling driver might register such a thread-creation callback to track the system-wide creation and deletion of threads against the driver's internal state. Windows 2000 registers up to eight such thread-creation callbacks. Any driver that successfully registers such a callback *must remain loaded until the system itself is shut down.*

After such a driver-supplied thread-creation routine is registered, it is called with *Create* set to TRUE whenever a new thread is created. The input *ThreadId* handle identifies the newly created thread. The input *ProcessId* handle identifies the process in which the given thread was just created. As each such thread is deleted, such a driver's thread-notify routine is called again with *Create* set to FALSE.

When it is called, the driver's thread-creation notify routine runs at IRQL PASSIVE_LEVEL either in the context of the newly created thread or in the context of the exiting thread.

Callers of **PsSetCreateThreadNotifyRoutine** must be running at IRQL PASSIVE_LEVEL.

See Also

PsGetCurrentProcessId, **PsGetCurrentThreadId**, **PsSetCreateProcessNotifyRoutine**, **PsSetLoadImageNotifyRoutine**

PsSetLoadImageNotifyRoutine

```
NTSTATUS
  PsSetLoadImageNotifyRoutine(
  IN PLOAD_IMAGE_NOTIFY_ROUTINE  NotifyRoutine
  );
```

PsSetLoadImageNotifyRoutine registers a driver-supplied callback that is subsequently notified whenever an image is loaded for execution.

Parameters

NotifyRoutine

Specifies the entry point of the caller-supplied load-image callback.

Include

ntddk.h

Return Value

PsSetLoadImageNotifyRoutine either returns STATUS_SUCCESS or it returns STATUS_INSUFFICIENT_RESOURCES if it failed the callback registration.

Comments

Highest-level system-profiling drivers can call **PsSetLoadImageNotifyRoutine** to set up their load-image notify routines, declared as follows:

```
VOID
(*PLOAD_IMAGE_NOTIFY_ROUTINE) (
    IN PUNICODE_STRING  FullImageName,
    IN HANDLE  ProcessId, // where image is mapped
    IN PIMAGE_INFO  ImageInfo
    );
```

After such a driver's callback has been registered, the system calls its load-image notify routine whenever an executable image is mapped into virtual memory, whether in system space or user space, before the execution of the image begins. Windows 2000 registers up to eight such load-image callbacks. Any driver that successfully registers such a callback *must remain loaded until the system itself is shut down.*

When the load-image notify routine is called, the input *FullImageName* points to a buffered Unicode string identifying the executable image file. The *ProcessId* handle identifies the process in which the image has been mapped, but this handle is zero if the newly loading image is a driver. The buffered data at *ImageInfo* is formatted as follows:

```
typedef struct  _IMAGE_INFO {
    union {
        ULONG  Properties;
        struct {
            ULONG ImageAddressingMode  : 8; //code addressing mode
            ULONG SystemModeImage      : 1; //system mode image
            ULONG ImageMappedToAllPids : 1; //mapped in all processes
            ULONG Reserved             : 22;
        };
    };
    PVOID  ImageBase;
    ULONG  ImageSelector;
    ULONG  ImageSize;
    ULONG  ImageSectionNumber;
} IMAGE_INFO, *PIMAGE_INFO;
```

When such a profiling driver's load-image routine is called, the members of this structure contain the following information:

ImageAddressingMode

Always set to IMAGE_ADDRESSING_MODE_32BIT.

SystemModeImage

Set either to one for newly loaded kernel-mode components, such as drivers, or to zero for images that are mapped into user space.

ImageMappedToAllPids and Reserved

Always set to zero.

ImageBase

Set to the virtual base address of the image.

ImageSelector

Always set to zero.

ImageSize

Set to the virtual size, in bytes, of the image.

ImageSectionNumber

Always set to zero.

Callers of **PsSetLoadImageNotifyRoutine** must be running at IRQL PASSIVE_LEVEL.

See Also

PsGetCurrentProcessId, **PsSetCreateProcessNotifyRoutine**, **PsSetCreateThread-NotifyRoutine**

PsTerminateSystemThread

```
NTSTATUS
  PsTerminateSystemThread(
  IN NTSTATUS  ExitStatus
  );
```

PsTerminateSystemThread terminates a caller-created system thread.

Parameters

ExitStatus

Specifies the status of the terminating system thread to the thread creator.

Include

wdm.h or *ntddk.h*

Return Value

PsTerminateSystemThread returns the STATUS_*XXX* supplied by the caller-created thread, usually STATUS_SUCCESS.

Comments

Drivers that create a device-dedicated thread call this routine, either when the driver is unloaded or when there are no outstanding I/O requests for the driver to process. For such a driver, **PsTerminateSystemThread** must be called in the context of the driver's thread; that is, the driver-created thread must terminate itself by making this call.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

PsCreateSystemThread

CHAPTER 9

Run-time Library Routines

References for these routines are in alphabetical order.

For an overview of the functionality of these routines, see Chapter 1, *Summary of Kernel-Mode Support Routines*.

InitializeListHead

```
VOID
  InitializeListHead(
  IN PLIST_ENTRY  ListHead
  );
```

InitializeListHead initializes any doubly linked, driver-managed interlocked queue or driver-maintained doubly linked list.

Parameters

ListHead

Points to the driver-allocated storage for the head of the interlocked queue or list. For an interlocked queue, the storage must be resident and the driver also must provide storage for a spin lock.

Include

wdm.h or *ntddk.h*

Comments

A driver that sets up internal interlocked queues for IRPs or manages internal linked lists must call **InitializeListHead**. A PnP driver should make the call from its AddDevice routine after creating the relevant device object; other drivers can call from the **DriverEntry** routine.

The *ListHead* of type LIST_ENTRY is doubly linked. Entries in an interlocked queue can be queued and dequeued by calling **ExInterlockedInsert..List** and **ExInterlockedRemoveHeadList**. Entries can be inserted into and removed from a driver-maintained list with **Insert..List** and **Remove..List**.

For an interlocked queue, a driver must provide resident storage: in the device extension of a driver-created device object, in the controller extension of a driver-created controller object, or in nonpaged pool allocated by the driver. The driver also must provide storage for a spin lock, which must be initialized with **KeInitializeSpinLock** before the driver's initial call to **ExInterlocked***Xxx* with the spin lock.

For a driver-maintained list, the driver must synchronize access to the list so that it is impossible for any two routines to be inserting and/or removing entries from the list simultaneously in SMP machines. Consequently, most drivers use the **ExInterlocked***Xxx* routines to manage the necessary synchronization, rather than setting up a driver-managed list, which is likely to require spin lock protection anyway.

Callers of **InitializeListHead** can be running at IRQL >= DISPATCH_LEVEL only if the caller-allocated storage for *ListHead* is resident.

See Also

ExInterlockedInsertHeadList, **ExInterlockedInsertTailList**, **ExInterlockedRemoveHeadList**, **ExInterlockedPopEntryList**, **ExInterlockedPushEntryList**, **InsertHeadList**, **InsertTailList**, **IsListEmpty**, **KeInitializeSpinLock**, **PopEntryList**, **PushEntryList**, **RemoveEntryList**, **RemoveHeadList**, **RemoveTailList**

InitializeObjectAttributes

```
VOID
  InitializeObjectAttributes(
  OUT POBJECT_ATTRIBUTES  InitializedAttributes,
  IN PUNICODE_STRING  ObjectName,
  IN ULONG  Attributes,
  IN HANDLE  RootDirectory,
  IN PSECURITY_DESCRIPTOR  SecurityDescriptor
  );
```

InitializeObjectAttributes sets up a parameter of type OBJECT_ATTRIBUTES for a subsequent call to **ExCreateCallback** or to a **ZwCreate***Xxx* or **ZwOpen***Xxx* routine.

Parameters

InitializedAttributes

Is a pointer to the initialized attributes for the object on return from **InitializeObjectAttributes**.

ObjectName

Is the full path name for the object.

Attributes

Is the set of attributes for the object, which can be a combination (ORed) of the following system-defined values:

OBJ_INHERIT
OBJ_PERMANENT
OBJ_EXCLUSIVE
OBJ_CASE_INSENSITIVE
OBJ_OPENIF

The validity of these values depends on the object type.

RootDirectory

Is a handle for the root directory in which the created object should be placed, in which an object to be opened is contained, or NULL.

SecurityDescriptor

Is an initialized absolute security descriptor for the object or NULL.

Include

wdm.h or *ntddk.h*

Comments

Callers of **InitializeObjectAttributes** must be running at IRQL PASSIVE_LEVEL.

See Also

ExCreateCallback, ***ZwCreateDirectoryObject***, **ZwCreateFile**, **ZwCreateKey**, **ZwOpenKey**, **ZwOpenSection**

InsertHeadList

```
VOID
  InsertHeadList(
  IN PLIST_ENTRY  ListHead,
  IN PLIST_ENTRY  Entry
  );
```

InsertHeadList inserts an entry at the head of a doubly linked, driver-managed list.

Parameters

ListHead

Points to the driver-allocated storage for the head of the list.

The *ListHead* of type LIST_ENTRY is doubly linked.

Entry

Points to an entry to be inserted in the list.

Include

wdm.h or *ntddk.h*

Comments

Callers of **InsertHeadList** can be running at IRQL >= DISPATCH_LEVEL only if the caller-allocated storage for *ListHead* is resident and only if pointers to every list entry remain valid at IRQL >= DISPATCH_LEVEL as well.

See Also

ExInterlockedInsertHeadList, **InitializeListHead**, **InsertTailList**, **IsListEmpty**, **RemoveHeadList**, **RemoveTailList**

InsertTailList

```
VOID
  InsertTailList(
  IN PLIST_ENTRY  ListHead,
  IN PLIST_ENTRY  Entry
  );
```

InsertTailList inserts an entry at the tail of a doubly linked, driver-managed list.

Parameters

ListHead

Points to the driver-allocated storage for the head of the list.

The *ListHead* of type LIST_ENTRY is doubly linked.

Entry

Points to an entry to be inserted in the list.

Include

wdm.h or *ntddk.h*

Comments

Callers of **InsertTailList** can be running at IRQL >= DISPATCH_LEVEL only if the caller-allocated storage for *ListHead* is resident and only if pointers to every list entry remain valid at IRQL >= DISPATCH_LEVEL as well.

See Also

ExInterlockedInsertTailList, **InitializeListHead**, **InsertHeadList**, **IsListEmpty**, **RemoveHeadList**, **RemoveTailList**

IsListEmpty

```
BOOLEAN
  IsListEmpty(
  IN PLIST_ENTRY  ListHead
  );
```

IsListEmpty indicates whether a doubly linked, driver-maintained list is empty.

Parameters

ListHead

Points to the driver-allocated storage for the head of the list.

The *ListHead* of type LIST_ENTRY is doubly linked.

Return Value

IsListEmpty returns TRUE if there are currently no entries in the list.

Include

wdm.h or *ntddk.h*

Comments

Callers of **IsListEmpty** can be running at IRQL >= DISPATCH_LEVEL only if the caller-allocated storage for *ListHead* is resident and only if pointers to every list entry remain valid at IRQL >= DISPATCH_LEVEL as well.

See Also

InitializeListHead, **RemoveHeadList**, **RemoveTailList**, **RemoveEntryList**

PopEntryList

```
PSINGLE_LIST_ENTRY
  PopEntryList(
  IN PSINGLE_LIST_ENTRY  ListHead
  );
```

PopEntryList removes an entry in a singly linked, driver-managed list.

Parameters

ListHead

Points to the driver-allocated storage for the head of the list.

The *ListHead* of type LIST_ENTRY is singly linked. The *ListHead* must be initialized to NULL before entries can be pushed and popped.

Include

wdm.h or *ntddk.h*

Return Value

PopEntryList returns a pointer to the last-pushed entry (LIFO order) or a NULL pointer if the list is currently empty.

Comments

Callers of **PopEntryList** can be running at IRQL >= DISPATCH_LEVEL only if the caller-allocated storage for *ListHead* is resident and only if pointers to every list entry remain valid at IRQL >= DISPATCH_LEVEL as well.

See Also

ExInterlockedPopEntryList, **PushEntryList**

PushEntryList

```
VOID
  PushEntryList(
  IN PSINGLE_LIST_ENTRY  ListHead,
  IN PSINGLE_LIST_ENTRY  Entry
  );
```

PushEntryList pushes an entry into a singly linked, driver-maintained list.

Parameters

ListHead

Points to the driver-allocated storage for the head of the list.

The *ListHead* of type SINGLE_LIST_ENTRY is singly linked. It must be initialized to NULL before the initial entry is pushed.

Entry

Points to the driver-allocated storage for an entry in the list.

Include

wdm.h or *ntddk.h*

Comments

Callers of **PushEntryList** can be running at IRQL >= DISPATCH_LEVEL only if the caller-allocated storage for *ListHead* is resident and only if pointers to every list entry remain valid at IRQL >= DISPATCH_LEVEL as well.

See Also

ExInterlockedPushEntryList, **PopEntryList**

RemoveEntryList

```
VOID
  RemoveEntryList(
  IN PLIST_ENTRY  Entry
  );
```

RemoveEntryList resets the links for an entry from a doubly linked, driver-managed list.

Parameters

Entry

Points to the entry.

Include

wdm.h or *ntddk.h*

Comments

Callers of **RemoveEntryList** can be running at IRQL >= DISPATCH_LEVEL only if the caller-allocated storage for *Entry* is resident.

RemoveEntryList sets the forward and backward links for the entry to each other.

See Also

InitializeListHead, IsListEmpty, RemoveHeadList, RemoveTailList

RemoveHeadList

```
PLIST_ENTRY
  RemoveHeadList(
  IN PLIST_ENTRY  ListHead
  );
```

RemoveHeadList removes an entry from the head of a doubly linked list.

Parameters

ListHead

Points to the driver-allocated storage for a doubly linked list with entries of type LIST_ENTRY.

Include

wdm.h or *ntddk.h*

Return Value

RemoveHeadList returns a pointer to the entry that was at the head of the list.

Comments

Calling **RemoveHeadList** with an empty list can cause a system failure. Callers of **RemoveHeadList** should first call **IsListEmpty** to determine if the list has any entries.

Callers of **RemoveHeadList** can be running at IRQL >= DISPATCH_LEVEL only if the caller-allocated storage for *ListHead* is resident and only if pointers to every list entry remain valid at IRQL >= DISPATCH_LEVEL as well.

This routine provides no inherent synchronization for the LIST_ENTRY that is being removed from the list.

See Also

ExInterlockedRemoveHeadList, InitializeListHead, IsListEmpty, RemoveTailList, RemoveEntryList

RemoveTailList

```
PLIST_ENTRY
  RemoveTailList(
  IN PLIST_ENTRY  ListHead
  );
```

RemoveTailList removes an entry from the tail of a doubly linked list.

Parameters

ListHead

Points to the driver-allocated storage for a doubly linked list with entries of type LIST_ENTRY.

Include

wdm.h or *ntddk.h*

Return Value

RemoveTailList returns a pointer to the entry that was at the tail of the list.

Comments

Calling **RemoveTailList** with an empty list can cause a system failure. Callers of **RemoveTailList** should first call **IsListEmpty** to determine if the list has any entries.

Callers of **RemoveTailList** can be running at IRQL >= DISPATCH_LEVEL only if the caller-allocated storage for *ListHead* is resident and only if pointers to every list entry remain valid at IRQL >= DISPATCH_LEVEL as well.

See Also

InitializeListHead, **IsListEmpty**, **RemoveHeadList**, **RemoveEntryList**

RtlAnsiStringToUnicodeSize

```
ULONG
  RtlAnsiStringToUnicodeSize(
  IN PANSI_STRING  AnsiString
  );
```

RtlAnsiStringToUnicodeSize returns the number of bytes required to hold an ANSI string converted into a Unicode string.

Parameters

AnsiString

Points to a buffer containing the ANSI string.

Return Value

RtlAnsiStringToUnicodeSize returns the necessary size in bytes for a Unicode string buffer.

Include

wdm.h or *ntddk.h*

Comments

Callers of **RtlAnsiStringToUnicodeSize** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlAnsiStringToUnicodeString

RtlAnsiStringToUnicodeString

```
NTSTATUS
  RtlAnsiStringToUnicodeString(
  IN OUT PUNICODE_STRING  DestinationString,
  IN PANSI_STRING  SourceString,
  IN BOOLEAN  AllocateDestinationString
  );
```

RtlAnsiStringToUnicodeString converts the given ANSI source string into a Unicode string. The translation conforms to the current system locale information.

Parameters

DestinationString

Points to a caller-allocated buffer for a converted Unicode string. If *AllocateDestination-String* is FALSE, the caller must also allocate a buffer for the **Buffer** member of *DestinationString* to hold the Unicode data. If *AllocateDestinationString* is TRUE, then **RtlAnsiStringToUnicodeString** allocates a Unicode data buffer large enough to hold the string, and passes a pointer to it in **Buffer**, and updates the length and maximum length members of *DestinationString* accordingly.

SourceString

Points to the ANSI string to be converted to Unicode.

AllocateDestinationString

Is TRUE if this routine should allocate the buffer space for the destination string. If it does, the caller must deallocate the buffer by calling **RtlFreeUnicodeString**.

Include

wdm.h or *ntddk.h*

Return Value

If the conversion succeeds, **RtlAnsiStringToUnicodeString** returns STATUS_SUCCESS. On failure, the routine does not allocate any memory.

Comments

If caller sets *AllocateDestinationString* to TRUE, the routine replaces the **Buffer** member of *DestinationString* with a pointer to the buffer it allocates. The old value can be overwritten even when the routine returns an error status code.

Callers of **RtlAnsiStringToUnicodeString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlAnsiStringToUnicodeSize, **RtlFreeUnicodeString**, **RtlInitAnsiString**, **RtlUnicodeStringToAnsiString**

RtlAppendUnicodeStringToString

```
NTSTATUS
  RtlAppendUnicodeStringToString(
  IN OUT PUNICODE_STRING  Destination,
  IN PUNICODE_STRING  Source
  );
```

RtlAppendUnicodeStringToString concatenates two Unicode strings. It copies bytes from the source up to the length of the destination buffer.

Parameters

Destination

Points to a buffered Unicode string.

Source

Points to the buffered string to be concatenated.

Include

wdm.h or *ntddk.h*

Return Value

RtlAppendUnicodeStringToString can return one of the following:

STATUS_SUCCESS

The source string was successfully appended to the destination counted string. The destination string length is updated to include the appended bytes.

STATUS_BUFFER_TOO_SMALL

The destination string length is too small to allow the source string to be concatenated. Accordingly, the destination string length is not updated.

Comments

The *Destination* and *Source* buffers must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

See Also

RtlAppendUnicodeToString

RtlAppendUnicodeToString

```
NTSTATUS
  RtlAppendUnicodeToString(
  IN  OUT PUNICODE_STRING  Destination,
  IN PCWSTR  Source
  );
```

RtlAppendUnicodeToString concatenates the supplied Unicode string to a buffered Unicode string. It copies bytes from the source string to the destination string up to the end of the destination buffer.

Parameters

Destination

Points to the buffered string.

Source

Points to the string to be appended to the *Destination* string.

Include

wdm.h or *ntddk.h*

Return Value

RtlAppendUnicodeToString can return one of the following:

STATUS_SUCCESS

The source string was successfully appended to the destination counted string. The destination string length is updated to include the appended bytes.

STATUS_BUFFER_TOO_SMALL

The destination string length was too small to allow the source string to be appended, so the destination string length is not updated.

Comments

The *Destination* buffer must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

See Also

RtlAppendUnicodeStringToString

RtlAreBitsClear

```
BOOLEAN
  RtlAreBitsClear(
  IN PRTL_BITMAP  BitMapHeader,
  IN ULONG  StartingIndex,
  IN ULONG  Length
  );
```

RtlAreBitsClear determines whether a given range of bits within a bitmap variable is clear.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

StartingIndex

Specifies the start of the bit range to be tested. This is a zero-based value indicating the position of the first bit in the range.

Length

Specifies how many bits to test.

Include

ntddk.h

Return Value

RtlAreBitsClear returns TRUE if *Length* consecutive bits beginning at *StartingIndex* are clear (that is, all the bits from *StartingIndex* to (*StartingIndex* + *Length*) -1). It returns FALSE if any bit in the given range is set, if the given range is not a proper subset of the bitmap, or if the given *Length* is zero.

Comments

Callers of **RtlAreBitsClear** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlAreBitsClear** can be called at any IRQL.

See Also

RtlAreBitsSet, **RtlCheckBit**, **RtlClearAllBits**, **RtlFindClearBits**, **RtlFindFirstRunClear**, **RtlFindLastBackwardRunClear**, **RtlFindLongestRunClear**, **RtlFindNextForwardRunClear**, **RtlInitializeBitMap**

RtlAreBitsSet

```
BOOLEAN
  RtlAreBitsSet(
  IN PRTL_BITMAP  BitMapHeader,
  IN ULONG  StartingIndex,
  IN ULONG  Length
  );
```

RtlAreBitsSet determines whether a given range of bits within a bitmap variable is set.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

StartingIndex

Specifies the start of the bit range to be tested. This is a zero-based value indicating the position of the first bit in the range.

Length

Specifies how many bits to test.

Include

ntddk.h

Return Value

RtlAreBitsSet returns TRUE if *Length* consecutive bits beginning at *StartingIndex* are set (that is, all the bits from *StartingIndex* to (*StartingIndex* + *Length*) –1). It returns FALSE if any bit in the given range is clear, if the given range is not a proper subset of the bitmap, or if the given *Length* is zero.

Comments

Callers of **RtlAreBitsSet** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlAreBitsSet** can be called at any IRQL.

See Also

RtlAreBitsClear, **RtlCheckBit**, **RtlFindSetBits**, **RtlInitializeBitMap**, **RtlSetAllBits**

RtlCharToInteger

```
NTSTATUS
  RtlCharToInteger(
  IN PCSZ  String,
  IN ULONG  Base  OPTIONAL,
  IN OUT PULONG  Value
  );
```

RtlCharToInteger converts a single-byte character to an integer value in the specified base.

Parameters

String

Points to a zero-terminated, single-byte character string.

Base

Specifies decimal, binary, octal, or hexadecimal base.

Value

Points to a location to which the converted value is returned.

Include

ntddk.h

Return Value

RtlCharToInteger returns STATUS_SUCCESS if the given character is converted. Otherwise, it can return STATUS_INVALID_PARAMETER.

Comments

RtlCharToInteger converts ANSI alphanumeric characters.

Callers of **RtlCharToInteger** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlInitString, **RtlIntegerToUnicodeString**

RtlCheckBit

```
ULONG
  RtlCheckBit(
  IN PRTL_BITMAP  BitMapHeader,
  IN ULONG  BitPosition
  );
```

RtlCheckBit determines whether a particular bit in a given bitmap variable is clear or set.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

BitPosition

Specifies which bit to check. This is a zero-based value indicating the position of the bit to be tested.

Include

ntddk.h

Return Value

RtlCheckBit returns zero if the given bit is clear or one if the given bit is set.

Comments

Callers of **RtlCheckBit** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlCheckBit** can be called at any IRQL.

See Also

RtlAreBitsClear, **RtlAreBitsSet**, **RtlInitializeBitMap**, **RtlNumberOfClearBits**, **RtlNumberOfSetBits**

RtlCheckRegistryKey

```
NTSTATUS
  RtlCheckRegistryKey(
  IN ULONG  RelativeTo,
  IN PWSTR  Path
  );
```

RtlCheckRegistryKey checks for the existence of a given named key in the registry.

Parameters

RelativeTo

Specifies whether *Path* is an absolute registry path or is relative to a predefined key path as one of the following:

Value	Meaning
RTL_REGISTRY_ABSOLUTE	Path is an absolute registry path.
RTL_REGISTRY_SERVICES	Path is relative to **\Registry\Machine\System\CurrentControlSet\Services**.
RTL_REGISTRY_CONTROL	Path is relative to **\Registry\Machine\System\CurrentControlSet\Control**.
RTL_REGISTRY_WINDOWS_NT	Path is relative to **\Registry\Machine\Software\Microsoft\Windows NT\CurrentVersion**.
RTL_REGISTRY_DEVICEMAP	Path is relative to **\Registry\Machine\Hardware\DeviceMap**.
RTL_REGISTRY_USER	Path is relative to **\Registry\User\CurrentUser**.
RTL_REGISTRY_OPTIONAL	Specifies that the key referenced by this parameter and the *Path* parameter are optional.
RTL_REGISTRY_HANDLE	Specifies that the *Path* parameter is actually a registry handle to use. This value is optional.

Path

Specifies the registry path according to the *RelativeTo* value. If RTL_REGISTRY_HANDLE is set, *Path* is a handle to be used directly.

Include

ntddk.h

Return Value

If the given, named key exists in the registry along the given relative path, **RtlCheckRegistryKey** returns STATUS_SUCCESS.

Comments

Callers of **RtlCheckRegistryKey** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlQueryRegistryValues

RtlClearAllBits

```
VOID
  RtlClearAllBits(
  IN PRTL_BITMAP  BitMapHeader
  );
```

RtlClearAllBits sets all bits in a given bitmap variable to zero.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

Include

ntddk.h

Comments

Callers of **RtlClearAllBits** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlClearAllBits** can be called at any IRQL.

See Also

RtlAreBitsClear, RtlAreBitsSet, RtlClearBits, RtlFindSetBits, RtlFindSetBitsAndClear, RtlInitializeBitMap, RtlNumberOfSetBits

RtlClearBits

```
VOID
  RtlClearBits(
  IN PRTL_BITMAP  BitMapHeader,
  IN ULONG  StartingIndex,
  IN ULONG  NumberToClear
  );
```

RtlClearBits sets all bits in a given range of a given bitmap variable to zero.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

StartingIndex

Specifies the start of the bit range to be cleared. This is a zero-based value indicating the position of the first bit in the range.

NumberToClear

Specifies how many bits to clear.

Include

ntddk.h

Comments

RtlClearBits simply returns control if the input *NumberToClear* is zero. *StartingIndex* plus *NumberToClear* must be less than or equal to **sizeof**(*BitMapHeader*->**SizeOfBitMap**).

Callers of **RtlClearBits** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlClearBits** can be called at any IRQL.

See Also

RtlAreBitsSet, RtlClearAllBits, RtlFindSetBits, RtlFindSetBitsAndClear, RtlInitializeBitMap, RtlNumberOfSetBits

RtlCompareMemory

```
SIZE_T
  RtlCompareMemory(
  IN CONST VOID  *Source1,
  IN CONST VOID  *Source2,
  IN SIZE_T  Length
  );
```

RtlCompareMemory compares blocks of memory and returns the number of bytes that are equivalent.

Parameters

Source1

Points to a block of memory to compare.

Source2

Points to a block of memory to compare.

Length

Specifies the number of bytes to be compared.

Include

wdm.h or *ntddk.h*

Return Value

RtlCompareMemory returns the number of bytes that compare as equal. If all bytes compare as equal, the input *Length* is returned.

Comments

Callers of **RtlCompareMemory** can be running at any IRQL if both blocks of memory are resident.

RtlCompareString

```
LONG
  RtlCompareString(
  IN PSTRING  String1,
  IN PSTRING  String2,
  BOOLEAN  CaseInsensitive
  );
```

RtlCompareString compares two counted strings.

Parameters

String1

Points to the first string.

String2

Points to the second string.

CaseInSensitive

If TRUE, case should be ignored when doing the comparison.

Include

ntddk.h

Return Value

RtlCompareString returns a signed value that gives the results of the comparison:

Zero

String1 equals String2.

< Zero

String1 is less than String2.

> Zero

String1 is greater than String2.

Comments

Callers of **RtlCompareString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlCompareUnicodeString, **RtlEqualString**

RtlCompareUnicodeString

```
LONG
  RtlCompareUnicodeString(
  IN PUNICODE_STRING  String1,
  IN PUNICODE_STRING  String2,
  IN BOOLEAN  CaseInSensitive
  );
```

RtlCompareUnicodeString compares two Unicode strings.

Parameters

String1

Points to the first string.

String2

Points to the second string.

CaseInSensitive

If TRUE, case should be ignored when doing the comparison.

Include

wdm.h or *ntddk.h*

Return Value

RtlCompareUnicodeString returns a signed value that gives the results of the comparison:

Zero

String1 equals String2.

< Zero

String1 is less than String2.

> Zero

String1 is greater than String2.

Comments

Callers of **RtlCompareUnicodeString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlCompareString, **RtlEqualString**

RtlConvertLongToLargeInteger

```
LARGE_INTEGER
  RtlConvertLongToLargeInteger(
  IN LONG  SignedInteger
  );
```

RtlConvertLongToLargeInteger converts the input signed integer to a signed large integer.

Parameters

SignedInteger

Specifies an integer of type LONG.

Include

wdm.h or *ntddk.h*

Return Value

RtlConvertLongToLargeInteger returns the large integer result.

Comments

Callers of **RtlConvertLongToLargeInteger** can be running at any IRQL.

RtlConvertLongToLuid

```
LUID
  RtlConvertLongToLuid(
  LONG Long
  );
```

RtlConvertLongToLuid converts a long integer to a locally unique identifier (LUID), which is used by the system to represent a security privilege.

Parameters

Long

Specifies the long integer to convert.

Include

ntddk.h

Return Value

RtlConvertLongToLuid returns the converted LUID.

Comments

RtlConvertLongToLuid is used to convert a system-defined privilege value to the locally unique identifier (LUID) used by the system to represent that privilege. Drivers typically pass a LUID to **SeSinglePrivilegeCheck** which is usually called by network transport drivers.

Callers of **RtlConvertLongToLuid** can be running at any IRQL.

See Also

RtlConvertULongToLuid, RtlEqualLuid, SeSinglePrivilegeCheck

RtlConvertUlongToLargeInteger

```
LARGE_INTEGER
  RtlConvertUlongToLargeInteger(
  IN ULONG  UnsignedInteger
  );
```

RtlConvertUlongToLargeInteger converts the input unsigned integer to a signed large integer.

Parameters

UnsignedInteger

Is a value of type ULONG.

Include

wdm.h or *ntddk.h*

Return Value

RtlConvertUlongToLargeInteger returns the converted large integer.

Comments

Callers of **RtlConvertUlongToLargeInteger** can be running at any IRQL.

RtlConvertUlongToLuid

```
LUID
  RtlConvertUlongToLuid(
  ULONG Ulong
  );
```

RtlConvertUlongToLuid converts an unsigned long integer to a locally unique identifier (LUID), which is used by the system to represent a security privilege.

Parameters

Ulong

Specifies the unsigned long integer to convert.

Include

ntddk.h

Return Value

RtlConvertUlongToLuid returns the converted LUID.

Comments

RtlConvertUlongToLuid is used to convert a system-defined privilege value, passed as a ULONG, to a locally unique identifier (LUID) used by the system to represent that privilege. Drivers typically pass a LUID to **SeSinglePrivilegeCheck** which is usually called by network transport drivers.

Callers of **RtlConvertUlongToLuid** can be running at any IRQL.

See Also

RtlConvertLongToLuid, **RtlEqualLuid**, **SeSinglePrivilegeCheck**

RtlCopyBytes

```
VOID
  RtlCopyBytes(
  IN PVOID  Destination,
  IN CONST VOID  *Source,
  IN SIZE_T  Length
  );
```

RtlCopyBytes copies a given number of bytes from one location to another.

Parameters

Destination

Points to the destination where the bytes are to be copied.

Source

Points to the memory to be copied.

Length

Specifies the number of bytes to be copied.

Include

wdm.h or *ntddk.h*

Comments

The (*Source* + *Length*) can overlap the *Destination* range passed to **RtlCopyBytes**.

Callers of **RtlCopyBytes** can be running at any IRQL if both memory blocks are resident. Otherwise, callers must be running at IRQL<DISPATCH_LEVEL.

See Also

RtlFillMemory, **RtlMoveMemory**, **RtlZeroMemory**

RtlCopyMemory

```
VOID
  RtlCopyMemory(
  IN VOID UNALIGNED  *Destination,
  IN CONST VOID UNALIGNED  *Source,
  IN SIZE_T  Length
  );
```

RtlCopyMemory copies the contents of one buffer to another.

Parameters

Destination

Points to the destination of the move.

Source

Points to the memory to be copied.

Length

Specifies the number of bytes to be copied.

Include

wdm.h or *ntddk.h*

Comments

RtlCopyMemory runs faster than **RtlMoveMemory**. However, the (*Source* + *Length*) cannot overlap the *Destination* range passed in to **RtlCopyMemory**.

Callers of **RtlCopyMemory** can be running at any IRQL if both memory blocks are resident. Otherwise, the caller must be running at IRQL < DISPATCH_LEVEL.

See Also

RtlMoveMemory

RtlCopyMemory32

```
VOID
RtlCopyMemory32 (
  VOID UNALIGNED *Destination,
  CONST VOID UNALIGNED *Source,
  ULONG Length
  );
```

RtlCopyMemory32 copies the contents of one memory buffer to another non-overlapping memory buffer.

Parameters

Destination

Pointer to the memory buffer where the data is copied.

Source

Pointer to the memory buffer from which data is copied.

Length

Specifies the number of bytes to copy.

Include

ntddk.h or *wdm.h*

Comments

RtlCopyMemory32 is the same as **RtlCopyMemory**, except that it copies at most 32 bits at a time.

Callers of **RtlCopyMemory** can be running at any IRQL if both memory blocks are resident. Otherwise, the caller must be running at IRQL < DISPATCH_LEVEL.

See Also

RtlCopyMemory, **RtlMoveMemory**

RtlCopyString

```
VOID
  RtlCopyString(
  IN OUT PSTRING  DestinationString,
  IN PSTRING  SourceString  OPTIONAL
  );
```

RtlCopyString copies a source string to a destination string.

Parameters

DestinationString

Points to the destination string buffer.

SourceString

Points to the source string buffer.

Include

ntddk.h

Comments

The *DestinationString* **Length** is set to zero if no source string is supplied, but this does not affect the length of the *DestinationString* buffer. The **MaximumLength** and **Buffer** members of the *DestinationString* are not modified by this routine.

The number of bytes copied from the *SourceString* is either the length of *SourceString* or the maximum length of *DestinationString*, whichever is smaller.

The *DestinationString* and *SourceString* buffers must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

See Also

RtlCopyUnicodeString

RtlCopyUnicodeString

```
VOID
  RtlCopyUnicodeString(
  IN OUT PUNICODE_STRING  DestinationString,
  IN PUNICODE_STRING  SourceString
  );
```

RtlCopyUnicodeString copies a source string to a destination string.

Parameters

DestinationString

Points to the destination string buffer.

SourceString

Points to the source string buffer.

Include

wdm.h or *ntddk.h*

Comments

If the source string is longer than the destination string, this routine copies bytes from the source to the end of the destination buffer, effectively truncating the copied source string.

The *DestinationString* and *SourceString* buffers must be resident if the caller is running at IRQL >= DISPATCH_LEVEL.

See Also

RtlCopyString

RtlCreateRegistryKey

```
NTSTATUS
  RtlCreateRegistryKey(
  IN ULONG  RelativeTo,
  IN PWSTR  Path
  );
```

RtlCreateRegistryKey adds a key object in the registry along a given relative path.

Parameters

RelativeTo

Specifies whether *Path* is an absolute registry path or is relative to a predefined key path as one of the following:

Value	Meaning
RTL_REGISTRY_ABSOLUTE	Path is an absolute registry path.
RTL_REGISTRY_SERVICES	Path is relative to **\Registry\Machine\System\CurrentControlSet\Services**.
RTL_REGISTRY_CONTROL	Path is relative to **\Registry\Machine\System\CurrentControlSet\Control**.
RTL_REGISTRY_WINDOWS_NT	Path is relative to **\Registry\Machine\Software\Microsoft\Windows NT\ CurrentVersion**.
RTL_REGISTRY_DEVICEMAP	Path is relative to **\Registry\Machine\Hardware\DeviceMap**.
RTL_REGISTRY_USER	Path is relative to **\Registry\User\CurrentUser**.

Continued

Value	Meaning
RTL_REGISTRY_OPTIONAL	Specifies that the key referenced by this parameter and the *Path* parameter are optional.
RTL_REGISTRY_HANDLE	Specifies that the *Path* parameter is actually a registry handle to use. This value is optional.

Path

Specifies the registry path according to the *RelativeTo* value. If RTL_REGISTRY_HANDLE is set, *Path* is a handle to be used directly.

Include

ntddk.h

Return Value

RtlCreateRegistryKey returns STATUS_SUCCESS if the key is created.

Comments

Callers of **RtlCreateRegistryKey** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlCheckRegistryKey, **RtlDeleteRegistryValue**, **RtlQueryRegistryValues**, **RtlWriteRegistryValue**, **ZwEnumerateKey**, **ZwOpenKey**

RtlCreateSecurityDescriptor

```
NTSTATUS
  RtlCreateSecurityDescriptor(
  IN OUT PSECURITY_DESCRIPTOR  SecurityDescriptor,
  IN ULONG  Revision
  );
```

RtlCreateSecurityDescriptor initializes a new absolute-format security descriptor. On return, the security descriptor is initialized with no system ACL, no discretionary ACL, no owner, no primary group, and all control flags set to zero.

Parameters

SecurityDescriptor

Points to the buffer for the security descriptor to be initialized.

Revision

Specifies the revision level to assign to the security descriptor.

Include

ntddk.h

Return Value

RtlCreateSecurityDescriptor can return one of the following:

STATUS_SUCCESS

The call completed successfully.

STATUS_UNKNOWN_REVISION

The given *Revision* is not supported.

Comments

In effect, a successful call to this routine initializes a security descriptor without security constraints.

Callers of **RtlCreateSecurityDescriptor** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlLengthSecurityDescriptor, **RtlSetDaclSecurityDescriptor**, **RtlValidSecurity-Descriptor**

RtlDeleteRegistryValue

```
NTSTATUS
  RtlDeleteRegistryValue(
  IN ULONG  RelativeTo,
  IN PCWSTR  Path,
  IN PCWSTR  ValueName
  );
```

RtlDeleteRegistryValue removes the specified entry name and the associated values from the registry along the given relative path.

Parameters

RelativeTo

Specifies whether *Path* is an absolute registry path or is relative to a predefined key path as one of the following:

Value	Meaning
RTL_REGISTRY_ABSOLUTE	Path is an absolute registry path.
RTL_REGISTRY_SERVICES	Path is relative to **\Registry\Machine\System\CurrentControlSet\Services**.
RTL_REGISTRY_CONTROL	Path is relative to **\Registry\Machine\System\CurrentControlSet\Control**.
RTL_REGISTRY_WINDOWS_NT	Path is relative to **\Registry\Machine\Software\Microsoft\Windows NT\CurrentVersion**.
RTL_REGISTRY_DEVICEMAP	Path is relative to **\Registry\Machine\Hardware\DeviceMap**.
RTL_REGISTRY_USER	Path is relative to **\Registry\User\CurrentUser**.
RTL_REGISTRY_OPTIONAL	Specifies that the key referenced by this parameter and the *Path* parameter are optional.
RTL_REGISTRY_HANDLE	Specifies that the *Path* parameter is actually a registry handle to use. This value is optional.

Path

Specifies the registry path according to the *RelativeTo* value. If RTL_REGISTRY_HANDLE is set, *Path* is a handle to be used directly.

ValueName

Points to the value name to be removed from the registry.

Include

wdm.h or *ntddk.h*

Return Value

RtlDeleteRegistryValue returns STATUS_SUCCESS if the value entry was deleted.

Callers of **RtlDeleteRegistryValue** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlCheckRegistryKey, **RtlQueryRegistryValues**, **RtlWriteRegistryValue**, **ZwEnumerateKey**, **ZwOpenKey**

RtlEnlargedIntegerMultiply

```
LARGE_INTEGER
  RtlEnlargedIntegerMultiply(
  IN LONG  Multiplicand,
  IN LONG  Multiplier
  );
```

RtlEnlargedIntegerMultiply is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlEnlargedUnsignedDivide

```
ULONG
  RtlEnlargedUnsignedDivide(
  IN ULARGE_INTEGER  Dividend,
  IN ULONG  Divisor,
  IN OUT PULONG  Remainder
  );
```

RtlEnlargedUnsignedDivide is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlEnlargedUnsignedMultiply

```
LARGE_INTEGER
  RtlEnlargedUnsignedMultiply(
  IN ULONG  Multiplicand,
  IN ULONG  Multiplier
  );
```

RtlEnlargedUnsignedMultiply is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlEqualLuid

```
BOOLEAN
  RtlEqualLuid(
  LUID Luid1,
  LUID Luid2
  );
```

RtlEqualLuid compares two LUIDs.

Parameters

Luid1

Points to a LUID to compare.

Luid2

Points to a LUID to compare.

Include

ntddk.h

Return Value

RtlEqualLuid returns TRUE if *Luid1* and *Luid2* are equivalent.

Comments

Callers of **RtlEqualLuid** can be running at any IRQL.

See Also

RtlConvertLongToLuid, **RtlConvertULongToLuid**, **SeSinglePrivilegeCheck**

RtlEqualMemory

```
ULONG
  RtlEqualMemory(
  CONST VOID *Source1,
  CONST VOID *Source2,
  SIZE_T Length
  );
```

RtlEqualMemory compares two blocks of memory to determine whether the specified number of bytes are identical.

Parameters

Source1

Points to a caller-allocated block of memory to compare.

Source2

Points to a caller-allocated block of memory that is compared to the block of memory to which *Source1* points.

Length

Specifies the number of bytes to be compared.

Include

wdm.h or *ntddk.h*

Return Value

RtlEqualMemory returns an unsigned value of one if *Source1* and *Source2* are equivalent; otherwise it returns zero.

Comments

RtlEqualMemory begins the comparison with byte zero of each block.

Callers of **RtlEqualMemory** can be running at any IRQL if both blocks of memory are resident.

See Also

RtlCompareMemory

RtlEqualString

```
BOOLEAN
  RtlEqualString(
  IN PSTRING  String1,
  IN PSTRING  String2,
  IN BOOLEAN  CaseInSensitive
  );
```

RtlEqualString compares two counted strings to determine whether they are equal.

Parameters

String1

Points to the first string.

String2

Points to the second string.

CaseInSensitive

If TRUE, case should be ignored when doing the comparison.

Include

ntddk.h

Return Value

RtlEqualString returns TRUE if the two strings are equal, otherwise it returns FALSE.

Comments

Callers of **RtlEqualString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlEqualUnicodeString

RtlEqualUnicodeString

```
BOOLEAN
  RtlEqualUnicodeString(
  IN CONST UNICODE_STRING  *String1,
  IN CONST UNICODE_STRING  *String2,
  IN BOOLEAN  CaseInSensitive
  );
```

RtlEqualUnicodeString compares two Unicode strings to determine whether they are equal.

Parameters

String1

Points to the first Unicode string.

String2

Points to the second Unicode string.

CaseInSensitive

If TRUE, case should be ignored when doing the comparison.

Include

wdm.h or *ntddk.h*

Return Value

RtlEqualUnicodeString returns TRUE if the two Unicode strings are equal.

Comments

Callers of **RtlEqualUnicodeString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlEqualString

RtlExtendedIntegerMultiply

```
LARGE_INTEGER
  RtlExtendedIntegerMultiply(
  IN LARGE_INTEGER  Multiplicand,
  IN LONG  Multiplier
  );
```

RtlExtendedIntegerMultiply is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlExtendedLargeIntegerDivide

```
LARGE_INTEGER
  RtlExtendedLargeIntegerDivide(
  IN LARGE_INTEGER  Dividend,
  IN ULONG  Divisor,
  IN OUT PULONG  Remainder
  );
```

RtlExtendedLargeIntegerDivide is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlExtendedMagicDivide

```
LARGE_INTEGER
  RtlExtendedMagicDivide(
  IN LARGE_INTEGER  Dividend,
  IN LARGE_INTEGER  MagicDivisor,
  IN CCHAR  ShiftCount
  );
```

RtlExtendedMagicDivide is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlFillBytes

```
VOID
  RtlFillBytes(
  PVOID Destination,
  SIZE_T Length,
  UCHAR Fill
  );
```

RtlFillBytes fills a caller-supplied buffer with the given unsigned character. For better performance, use **RtlFillMemory**.

RtlFillMemory

```
VOID
  RtlFillMemory(
  IN VOID UNALIGNED  *Destination,
  IN SIZE_T  Length,
  IN UCHAR  Fill
  );
```

RtlFillMemory fills a caller-supplied buffer with the given character.

Parameters

Destination

Points to the memory to be filled.

Length

Specifies the number of bytes to be filled.

Fill

Specifies the value to fill the memory.

Include

wdm.h or *ntddk.h*

Comments

Callers of **RtlFillMemory** can be running at any IRQL, provided that the *Destination* buffer is resident.

See Also

RtlZeroMemory

RtlFindClearBits

```
ULONG
  RtlFindClearBits(
  IN PRTL_BITMAP  BitMapHeader,
  IN ULONG  NumberToFind,
  IN ULONG  HintIndex
  );
```

RtlFindClearBits searches for a range of clear bits of a requested size within a bitmap.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

NumberToFind

Specifies how many contiguous clear bits will satisfy this request.

HintIndex

Specifies a zero-based bit position around which to start looking for a clear bit range of the given size.

Include

ntddk.h

Return Value

RtlFindClearBits either returns the zero-based starting bit index for a clear bit range of at least the requested size, or it returns 0xFFFFFFFF if it cannot find such a range within the given bitmap variable.

Comments

For a successful call, the returned bit position is not necessarily equivalent to the given *HintIndex*. If necessary, **RtlFindClearBits** searches the whole bitmap to locate a clear bit range of the requested size. However, it starts searching for the requested range near *HintIndex*, so callers can find such a range more quickly when they can supply appropriate hints about where to start looking.

Callers of **RtlFindClearBits** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlFindClearBits** can be called at any IRQL.

See Also

RtlAreBitsClear, **RtlFindClearBitsAndSet**, **RtlFindFirstRunClear**, **RtlFindLongestRunClear**, **RtlInitializeBitMap**, **RtlNumberOfClearBits**, **RtlFindSetBits**

RtlFindClearBitsAndSet

```
ULONG
  RtlFindClearBitsAndSet(
  IN PRTL_BITMAP  BitMapHeader,
  IN ULONG  NumberToFind,
  IN ULONG  HintIndex
  );
```

RtlFindClearBitsAndSet searches for a range of clear bits of a requested size within a bitmap and sets all bits in the range when it has been located.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

NumberToFind

Specifies how many contiguous clear bits will satisfy this request.

HintIndex

Specifies a zero-based bit position around which to start looking for a clear bit range of the given size.

Include

ntddk.h

Return Value

RtlFindClearBitsAndSet either returns the zero-based starting bit index for a clear bit range of the requested size that it set, or it returns 0xFFFFFFFF if it cannot find such a range within the given bitmap variable.

Comments

For a successful call, the returned bit position is not necessarily equivalent to the given *HintIndex*. If necessary, **RtlFindClearBitsAndSet** searches the whole bitmap to locate a clear bit range of the requested size. However, it starts searching for the requested range near *HintIndex*, so callers can have such a range reset more quickly when they can supply appropriate hints about where to start looking.

Callers of **RtlFindClearBitsAndSet** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlFindClearBitsAndSet** can be called at any IRQL.

See Also

RtlAreBitsClear, RtlFindClearBits, RtlFindClearRuns, RtlFindFirstRunClear, RtlFindLastBackwardRunClear, RtlFindLongestRunClear, RtlFindNextForwardRunClear, RtlInitializeBitMap, RtlNumberOfClearBits, RtlSetAllBits, RtlSetBits

RtlFindClearRuns

```
ULONG
  RtlFindClearRuns(
  IN PRTL_BITMAP BitMapHeader,
  OUT PRTL_BITMAP_RUN RunArray,
  IN ULONG SizeOfRunArray,
  IN BOOLEAN LocateLongestRuns
  );
```

RtlFindClearRuns finds the specified number of runs of clear bits within a given bitmap variable.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

RunArray

Points to the first element in a caller-allocated array for the bit position and length of each clear run found in the given bitmap variable.

SizeOfRunArray

Specifies the maximum number of clear runs to satisfy this request.

LocateLongestRuns

If TRUE, specifies that the routine is to search the entire bitmap for the longest clear runs it can find. Otherwise, the routine stops searching when it has found the number of clear runs specified by *SizeOfRunArray*.

Include

ntddk.h

Return Value

RtlFindClearRuns returns the number of clear runs found.

Comments

If *LocateLongestRuns* is TRUE, the clear runs indicated at *RunArray* are sorted from longest to shortest. A clear run can consist of a single bit.

Callers of **RtlFindClearRuns** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlFindClearRuns** can be called at any IRQL.

See Also

RtlAreBitsClear, RtlFindClearBits, RtlFindLongestRunClear, RtlFindFirstRunClear, RtlFindNextForwardRunClear, RtlFindLastBackwardRunClear, RtlInitializeBitMap

RtlFindFirstRunClear

```
ULONG
  RtlFindFirstRunClear(
  IN PRTL_BITMAP  BitMapHeader,
  OUT PULONG  StartingIndex
  );
```

RtlFindFirstRunClear searches for the initial contiguous range of clear bits within a given bitmap variable.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

StartingIndex

Points to a variable in which the starting index of the initial clear run in the bitmap is returned. This is a zero-based value indicating the bit position of the first clear bit in the returned range. Its value is meaningless if **RtlFindFirstRunClear** cannot find a run of clear bits.

Include

ntddk.h

Return Value

RtlFindFirstRunClear returns either the number of bits in the run beginning at *StartingIndex* or zero if it cannot find a run of clear bits within the bitmap.

Comments

RtlFindFirstRunClear sets *StartingIndex* to 0xFFFFFFFF if it cannot find a contiguous range of clear bits within the given bitmap variable. A returned run can have a single clear bit. That is, once a clear bit is found, **RtlFindFirstRunClear** continues searching until it finds the next set bit and, then, returns the number of clear bits in the run it found.

Callers of **RtlFindFirstRunClear** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlFindFirstRunClear** can be called at any IRQL.

See Also

RtlAreBitsClear, RtlFindClearBits, RtlFindClearRuns, RtlFindFirstRunClear, RtlFindLastBackwardRunClear, RtlFindLongestRunClear, RtlFindNextForwardRunClear, RtlSetBits, RtlInitializeBitMap, RtlNumberOfClearBits

RtlFindLastBackwardRunClear

```
ULONG
  RtlFindLastBackwardRunClear(
  IN PRTL_BITMAP BitMapHeader,
  IN ULONG FromIndex,
  OUT PULONG StartingRunIndex
  );
```

RtlFindLastBackwardRunClear searches a given bitmap variable for the preceding clear run of bits, starting from the specified index position.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

FromIndex

Specifies a zero-based bit position at which to start looking for a clear run of bits.

StartingRunIndex

Points to a variable in which the starting index of the clear run found in the bitmap is returned. This is a zero-based value indicating the bit position of the first clear bit in the run preceding the given *FromIndex*. Its value is meaningless if **RtlFindLastBackwardRunClear** cannot find a run of clear bits.

Include

ntddk.h

Return Value

RtlFindLastBackwardRunClear returns the number of bits in the run beginning at *StartingRunIndex* or zero if it cannot find a run of clear bits preceding *FromIndex* in the bitmap.

Comments

Callers of **RtlFindLastBackwardRunClear** must be running at IRQL < DISPATCH_LEVEL.

See Also

RtlAreBitsClear, RtlFindClearBits, RtlFindClearRuns, RtlFindFirstRunClear, RtlFindLastBackwardRunClear, RtlFindLongestRunClear, RtlFindNextFoward-RunClear, RtlInitializeBitMap

RtlFindLeastSignificantBit

```
CCHAR
  RtlFindLeastSignificantBit(
  IN ULONGLONG Set
);
```

RtlFindLeastSignificantBit returns the zero-based position of the least significant non-zero bit in its parameter.

Parameters

Set

The 64-bit value to be searched for its least significant non-zero bit.

Include

ntddk.h

Return Value

The zero-based bit position of the least significant non-zero bit, or -1 if every bit is zero.

See Also

RtlFindMostSignificantBit

RtlFindMostSignificantBit

```
CCHAR
  RtlFindMostSignificantBit(
  IN ULONGLONG Set
);
```

RtlFindMostSignificantBit returns the zero-based position of the most significant non-zero bit in its parameter.

Parameters

Set

The 64-bit value to be searched for its most significant non-zero bit.

Include

ntddk.h

Return Value

The zero-based bit position of the most significant non-zero bit, or -1 if every bit is zero.

See Also

RtlFindLeastSignificantBit

RtlFindLongestRunClear

```
ULONG
  RtlFindLongestRunClear(
  IN PRTL_BITMAP  BitMapHeader,
  OUT PULONG  StartingIndex
  );
```

RtlFindLongestRunClear searches for the largest contiguous range of clear bits within a given bitmap variable.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

StartingIndex

Points to a variable in which the starting index of the longest clear run in the bitmap is returned. This is a zero-based value indicating the bit position of the first clear bit in the returned range.

Include

ntddk.h

Return Value

RtlFindLongestRunClear returns either the number of bits in the run beginning at *StartingIndex* or zero if it cannot find a run of clear bits within the bitmap.

Comments

RtlFindLongestRunClear sets *StartingIndex* to 0xFFFFFFFF if it cannot find a contiguous range of clear bits within the given bitmap variable. A returned run can have a single clear bit.

Callers of **RtlFindLongestRunClear** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlFindLongestRunClear** can be called at any IRQL.

See Also

RtlAreBitsClear, **RtlFindClearBits**, **RtlFindClearRuns**, **RtlFindFirstRunClear**, **RtlFindLastBackwardRunClear**, **RtlFindLongestRunClear**, **RtlFindNextForwardRunClear**, **RtlSetBits**, **RtlInitializeBitMap**, **RtlNumberOfClearBits**

RtlFindNextForwardRunClear

```
ULONG
  RtlFindNextForwardRunClear(
  IN PRTL_BITMAP BitMapHeader,
  IN ULONG FromIndex,
  OUT PULONG StartingRunIndex
  );
```

RtlFindNextForwardRunClear searches a given bitmap variable for the next clear run of bits, starting from the specified index position.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

FromIndex

Specifies a zero-based bit position at which to start looking for a clear run of bits.

StartingRunIndex

Points to a variable in which the starting index of the clear run found in the bitmap is returned. This is a zero-based value indicating the bit position of the first clear bit in the run. Its value is meaningless if **RtlFindNextForwardRunClear** cannot find a run of clear bits.

Include

ntddk.h

Return Value

RtlFindNextFowardRunClear returns either the number of bits in the run beginning at *StartingRunIndex* or zero if it cannot find a run of clear bits following *FromIndex* in the bitmap.

Comments

Callers of **RtlFindNextForwardRunClear** must be running at IRQL < DISPATCH_LEVEL.

See Also

RtlAreBitsClear, **RtlFindClearBits**, **RtlFindClearRuns**, **RtlFindFirstRunClear**, **RtlFindLastBackwardRunClear**, **RtlFindLongestRunClear**, **RtlInitializeBitMap**

RtlFindSetBits

```
ULONG
  RtlFindSetBits(
  IN PRTL_BITMAP  BitMapHeader,
  IN ULONG  NumberToFind,
  IN ULONG  HintIndex
  );
```

RtlFindSetBits searches for a range of set bits of a requested size within a bitmap.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

NumberToFind

Specifies how many contiguous set bits will satisfy this request.

HintIndex

Specifies a zero-based bit position around which to start looking for a set bit range of the given size.

Include

ntddk.h

Return Value

RtlFindSetBits either returns the zero-based starting bit index for a set bit range of the requested size, or it returns 0xFFFFFFFF if it cannot find such a range within the given bitmap variable.

Comments

For a successful call, the returned bit position is not necessarily equivalent to the given *HintIndex*. If necessary, **RtlFindSetBits** searches the whole bitmap to locate a set bit range of the requested size. However, it starts searching for the requested range near *HintIndex*, so callers can find such a range more quickly when they can supply appropriate hints about where to start looking.

Callers of **RtlFindSetBits** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlFindSetBits** can be called at any IRQL.

See Also

RtlAreBitsSet, **RtlClearBits**, **RtlFindClearBits**, **RtlFindSetBitsAndClear**, **RtlInitializeBitMap**, **RtlNumberOfSetBits**

RtlFindSetBitsAndClear

```
ULONG
  RtlFindSetBitsAndClear(
  IN PRTL_BITMAP  BitMapHeader,
  IN ULONG  NumberToFind,
  IN ULONG  HintIndex
  );
```

RtlFindSetBitsAndClear searches for a range of set bits of a requested size within a bitmap and clears all bits in the range when it has been located.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

NumberToFind

Specifies how many contiguous set bits will satisfy this request.

HintIndex

Specifies a zero-based bit position around which to start looking for a set bit range of the given size.

Include

ntddk.h

Return Value

RtlFindSetBitsAndClear either returns the zero-based starting bit index for a set bit range of the requested size that it cleared, or it returns 0xFFFFFFFF if it cannot find such a range within the given bitmap variable.

Comments

For a successful call, the returned bit position is not necessarily equivalent to the given *HintIndex*. If necessary, **RtlFindSetBitsAndClear** searches the whole bitmap to locate a set bit range of the requested size. However, it starts searching for the requested range near *HintIndex*, so callers can clear such a range more quickly when they can supply appropriate hints about where to start looking.

Callers of **RtlFindSetBitsAndClear** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlFindSetBitsAndClear** can be called at any IRQL.

See Also

RtlAreBitsSet, **RtlClearAllBits**, **RtlClearBits**, **RtlFindSetBits**, **RtlInitializeBitMap**, **RtlNumberOfSetBits**

RtlFreeAnsiString

```
VOID
  RtlFreeAnsiString(
  IN PANSI_STRING  AnsiString
  );
```

RtlFreeAnsiString releases storage that was allocated by **RtlUnicodeStringToAnsiString**.

Parameters

AnsiString

Points to the ANSI string buffer previously allocated by **RtlUnicodeStringToAnsiString**.

Include

wdm.h or *ntddk.h*

Comments

This routine does not release the Unicode string buffer passed to **RtlUnicodeStringToAnsiString**.

Callers of **RtlFreeAnsiString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlUnicodeStringToAnsiString

RtlFreeUnicodeString

```
VOID
  RtlFreeUnicodeString(
  IN PUNICODE_STRING  UnicodeString
  );
```

RtlFreeUnicodeString releases storage that was allocated by **RtlAnsiStringToUnicodeString** or **RtlUpcaseUnicodeString**.

Parameters

UnicodeString

Points to the Unicode string buffer previously allocated by **RtlAnsiStringToUnicodeString** or **RtlUpcaseUnicodeString**.

Include

wdm.h or *ntddk.h*

Comments

This routine does not release the ANSI string buffer passed to **RtlAnsiStringToUnicodeString**.

Callers of **RtlFreeUnicodeString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlAnsiStringToUnicodeString, RtlUpcaseUnicodeString

RtlGetVersion

```
NTSTATUS
RtlGetVersion(
  IN OUT PRTL_OSVERSIONINFOW lpVersionInformation
  );
```

RtlGetVersion returns version information about the currently running operating system.

Parameters

lpVersionInformation

Pointer to either an RTL_OSVERSIONINFOW or RTL_OSVERSIONINFOEXW structure that contains the version information about the currently running operating system. A caller specifies which input structure is used by setting the **dwOSVersionInfoSize** member of the structure to the size in bytes of the structure that is used.

Include

ntddk.h

Return Value

RtlGetVersion returns STATUS_SUCCESS.

Comments

RtlGetVersion is the kernel-mode equivalent of the user-mode **GetVersionEx** function in the Microsoft® Platform SDK. See the example in the Platform SDK that shows how to get the system version.

When using **RtlGetVersion** to determine whether a particular version of the operating system is running, a caller should check for version numbers that are greater than or equal to the required version number. This ensures that a version test succeeds for later versions of the operating system.

Because operating system features can be added in a redistributable DLL, checking only the major and minor version numbers is not the most reliable way to verify the presence of a specific system feature. A driver should use **RtlVerifyVersionInfo** to test for the presence of a specific system feature.

See Also

RTL_OSVERSIONINFOW, RTL_OSVERSIONINFOEXW, RtlVerifyVersionInfo

RtlGUIDFromString

```
NTSTATUS
  RtlGUIDFromString(
  IN PUNICODE_STRING GuidString,
  OUT GUID *Guid
  );
```

RtlGUIDFromString converts the given Unicode string to a GUID in binary format.

Parameters

GuidString

Points to the buffered Unicode string to be converted to a GUID.

Guid

Points to a caller-supplied variable in which the GUID is returned.

Include

wdm.h or *ntddk.h*

Return Value

If the conversion succeeds, **RtlGUIDFromString** returns STATUS_SUCCESS. Otherwise, no conversion was done.

Comments

Callers of **RtlGUIDFromString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlStringFromGUID

RtlInitAnsiString

```
VOID
  RtlInitAnsiString(
  IN OUT PANSI_STRING  DestinationString,
  IN PCSZ  SourceString
  );
```

RtlInitAnsiString initializes a counted ANSI string.

Parameters

DestinationString

Points to the buffer for a counted ANSI string to be initialized.

SourceString

Points to a zero-terminated string with which to initialize the counted string.

Include

wdm.h or *ntddk.h*

Comments

The *DestinationString* is initialized to point to the *SourceString*. The length and maximum length for the *DestinationString* are initialized to the length of the *SourceString*. If *SourceString* is NULL, the lengths are zero.

Callers of **RtlInitAnsiString** can be running at IRQL <= DISPATCH_LEVEL if the *DestinationString* buffer is nonpageable. Usually, callers run at IRQL PASSIVE_LEVEL because most other **Rtl..String** routines cannot be called at raised IRQL.

See Also

RtlInitString, **RtlInitUnicodeString**

RtlInitializeBitMap

```
VOID
  RtlInitializeBitMap(
  IN PRTL_BITMAP  BitMapHeader,
  IN PULONG  BitMapBuffer,
  IN ULONG  SizeOfBitMap
  );
```

RtlInitializeBitMap initializes the header of a bitmap variable.

Parameters

BitMapHeader

Points to the bitmap header for which the caller provides storage, which must be at least **sizeof**(RTL_BITMAP).

BitMapBuffer

Points to caller-allocated memory for the bitmap itself. The base address of this buffer must be ULONG-aligned.

SizeOfBitMap

Specifies the number of bits in the bitmap. This value must be an integral multiple of the number of bits in a ULONG.

Include

ntddk.h

Comments

A driver can use a bitmap variable as an economical way to keep track of a set of reusable items. For example, file systems use a bitmap variable to track which clusters/sectors on a disk have already been allocated to hold file data. The system-supplied SCSI port driver uses a bitmap variable to track which queue tags have been assigned to SRBs.

RtlInitializeBitMap must be called before any other **Rtl***Xxx* that operates on a bitmap variable. The *BitMapHeader* pointer is an input parameter in all subsequent calls to the **Rtl***Xxx* that operate on the caller's bitmap variable at *BitMapBuffer*. The caller is responsible for synchronizing access to the bitmap variable.

If an already initialized bitmap header is reinitialized, the subsequent call to **RtlInitialize-BitMap** has no effect on the current contents of the associated bitmap variable.

Callers of **RtlInitializeBitMap** must be running at IRQL < DISPATCH_LEVEL. Callers of **Rtl***Xxx* that operate on an initialized bitmap variable must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitmapHeader* is pageable; otherwise, callers can be running at any IRQL.

See Also

ExInitializeFastMutex, RtlAreBitsClear, RtlAreBitsSet, RtlCheckBit, RtlClear-AllBits, RtlClearBits, RtlFindClearBits, RtlFindClearBitsAndSet, RtlFindClearRuns, RtlFindFirstRunClear, RtlFindLastBackwardRunClear, RtlFindLongestRunClear, RtlFindNextForwardRunClear, RtlFindSetBits, RtlFindSetBitsAndClear, RtlNumber-OfClearBits, RtlNumberOfSetBits, RtlSetAllBits, RtlSetBits

RtlInitString

```
VOID
  RtlInitString(
  IN OUT PSTRING  DestinationString,
  IN PCSZ  SourceString
  );
```

RtlInitString initializes a counted string.

Parameters

DestinationString

Points to the buffer for a counted string to be initialized.

SourceString

Points to a NUL-terminated string value with which to initialize the counted string.

Include

wdm.h or *ntddk.h*

Comments

DestinationString is initialized to point to *SourceString* and the length and maximum length for the *DestinationString* are initialized to the length of *SourceString*. The lengths are is zero if *SourceString* is NULL.

Callers of **RtlInitString** can be running at IRQL <= DISPATCH_LEVEL if the *DestinationString* buffer is nonpageable. Usually, callers run at IRQL PASSIVE_LEVEL because most other **Rtl..String** routines cannot be called at raised IRQL.

See Also

RtlInitAnsiString, RtlInitUnicodeString

RtlInitUnicodeString

```
VOID
  RtlInitUnicodeString(
  IN OUT PUNICODE_STRING  DestinationString,
  IN PCWSTR  SourceString
  );
```

RtlInitUnicodeString initializes a counted Unicode string.

Parameters

DestinationString

Points to the buffer for a counted Unicode string to be initialized.

SourceString

Points to a zero-terminated Unicode string with which to initialize the counted string.

Include

wdm.h or *ntddk.h*

Comments

DestinationString is initialized to point to *SourceString*. The length and maximum length for *DestinationString* are initialized to the length of *SourceString*. If *SourceString* is NULL, the length is zero.

Callers of **RtlInitUnicodeString** can be running at IRQL <= DISPATCH_LEVEL if the *DestinationString* buffer is nonpageable. Usually, callers run at IRQL PASSIVE_LEVEL because most other **Rtl..String** routines cannot be called at raised IRQL.

See Also

RtlInitAnsiString, RtlInitString

RtlInt64ToUnicodeString

```
NTSTATUS
RtlInt64ToUnicodeString (
  IN ULONGLONG Value,
  IN ULONG Base OPTIONAL,
  IN OUT PUNICODE_STRING String
  );
```

RtlInt64ToUnicodeString converts a specified unsigned 64-bit integer value to a Unicode string that represents the value in a specified base.

Parameters

Value

Specifies a 64-bit unsigned integer.

Base

Specifies the following base for the conversion as follows:

Value	Base
16	Hexadecimal
8	Octal
2	Binary
0 or 10	Decimal

String

Pointer to a Unicode string that represents the value of *Value*. The caller must allocate the Unicode string buffer.

Include

ntddk.h or *wdm.h*

Return Value

RtlInt64ToUnicodeString returns one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_OVERFLOW

The Unicode string buffer is too small.

STATUS_INVALID_PARAMETER

The specified code base is not valid.

Comments

Callers of **RtlInt64ToUnicodeString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlAppendUnicodeStringToString, **RtlUnicodeStringToInteger**

RtlIntegerToUnicodeString

```
NTSTATUS
  RtlIntegerToUnicodeString(
  IN ULONG  Value,
  IN ULONG  Base  OPTIONAL,
  IN OUT PUNICODE_STRING  String
  );
```

RtlIntegerToUnicodeString converts a given unsigned integer value in the specified base to one or more buffered Unicode characters.

Parameters

Value

Identifies an unsigned integer of type ULONG.

Base

Specifies decimal, binary, octal, or hexadecimal base.

String

Points to a buffer to contain the converted value.

Include

wdm.h or *ntddk.h*

Return Value

If **RtlIntegerToUnicodeString** succeeds, it returns STATUS_SUCCESS. Otherwise, it can return STATUS_INVALID_PARAMETER if the given *Value* is illegal for the specified base.

Comments

Callers of **RtlIntegerToUnicodeString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlUnicodeStringToInteger, **RtlAppendUnicodeStringToString**

RtlIntPtrToUnicodeString

```
NTSTATUS
RtlIntPtrToUnicodeString (
  PLONG Value,
  ULONG Base OPTIONAL,
  PUNICODE_STRING String
  );
```

RtlIntPtrToUnicodeString converts a specified PLONG value to a Unicode string that represents the pointer value in a specified base.

Parameters

Value

Specifies an integer pointer.

Base

Specifies the following base for the conversion:

Value	Base
16	Hexadecimal
8	Octal
2	Binary
0 or 10	Decimal

String

Pointer to a Unicode string that represents the value of *Value*. The caller must allocate the Unicode string buffer.

Include

ntddk.h or *wdm.h*

Comments

Callers of **RtlIntPtrToUnicodeString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlAppendUnicodeStringToString, RtlIntegerToUnicodeString, RtlUnicodeStringToInteger

RtlLargeIntegerAdd

```
LARGE_INTEGER
  RtlLargeIntegerAdd(
  IN LARGE_INTEGER  Addend1,
  IN LARGE_INTEGER  Addend2
  );
```

RtlLargeIntegerAdd is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerAnd

```
VOID
  RtlLargeIntegerAnd(
  IN OUT LARGE_INTEGER  Result,
  IN LARGE_INTEGER  Source,
  IN LARGE_INTEGER  Mask
  );
```

RtlLargeIntegerAnd is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerArithmeticShift

```
LARGE_INTEGER
  RtlLargeIntegerArithmeticShift(
  IN LARGE_INTEGER  LargeInteger,
  IN CCHAR  ShiftCount
  );
```

RtlLargeIntegerArithmeticShift is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerDivide

```
LARGE_INTEGER
  RtlLargeIntegerDivide(
  IN LARGE_INTEGER  Dividend,
  IN LARGE_INTEGER  Divisor,
  IN OUT PLARGE_INTEGER  Remainder
  );
```

RtlLargeIntegerDivide is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerEqualTo

```
BOOLEAN
  RtlLargeIntegerEqualTo(
  IN LARGE_INTEGER  Operand1,
  IN LARGE_INTEGER  Operand2
  );
```

RtlLargeIntegerEqualTo is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerEqualToZero

```
BOOLEAN
  RtlLargeIntegerEqualToZero(
  IN LARGE_INTEGER  Operand
  );
```

RtlLargeIntegerEqualToZero is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerGreaterThan

```
BOOLEAN
  RtlLargeIntegerGreaterThan(
  IN LARGE_INTEGER  Operand1,
  IN LARGE_INTEGER  Operand2
  );
```

RtlLargeIntegerGreaterThan is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerGreaterThanOrEqualTo

```
BOOLEAN
  RtlLargeIntegerGreaterThanOrEqualTo(
  IN LARGE_INTEGER  Operand1,
  IN LARGE_INTEGER  Operand2
  );
```

RtlLargeIntegerGreaterThanOrEqualTo is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerGreaterOrEqualToZero

```
BOOLEAN
  RtlLargeIntegerGreaterOrEqualToZero(
  IN LARGE_INTEGER  Operand
  );
```

RtlLargeIntegerGreaterOrEqualToZero is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerGreaterThanZero

```
BOOLEAN
  RtlLargeIntegerGreaterThanZero(
  IN LARGE_INTEGER  Operand
  );
```

RtlLargeIntegerGreaterThanZero is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerLessThan

```
BOOLEAN
  RtlLargeIntegerLessThan(
  IN LARGE_INTEGER  Operand1,
  IN LARGE_INTEGER  Operand2
  );
```

RtlLargeIntegerLessThan is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerLessThanOrEqualTo

```
BOOLEAN
  RtlLargeIntegerLessThanOrEqualTo(
  IN LARGE_INTEGER  Operand1,
  IN LARGE_INTEGER  Operand2
  );
```

RtlLargeIntegerLessThanOrEqualTo is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerLessOrEqualToZero

```
BOOLEAN
  RtlLargeIntegerLessOrEqualToZero(
  IN LARGE_INTEGER  Operand
  );
```

RtlLargeIntegerLessOrEqualToZero is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerLessThanZero

```
BOOLEAN
  RtlLargeIntegerLessThanZero(
  IN LARGE_INTEGER  Operand
  );
```

RtlLargeIntegerLessThanZero is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerNegate

```
LARGE_INTEGER
  RtlLargeIntegerNegate(
  IN LARGE_INTEGER  Subtrahend
  );
```

RtlLargeIntegerNegate is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerNotEqualTo

```
BOOLEAN
  RtlLargeIntegerNotEqualTo(
  IN LARGE_INTEGER  Operand1,
  IN LARGE_INTEGER  Operand2
  );
```

RtlLargeIntegerNotEqualTo is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerNotEqualToZero

```
BOOLEAN
  RtlLargeIntegerNotEqualToZero(
  IN LARGE_INTEGER  Operand
  );
```

RtlLargeIntegerNotEqualToZero is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerShiftLeft

```
LARGE_INTEGER
  RtlLargeIntegerShiftLeft(
  IN LARGE_INTEGER  LargeInteger,
  IN CCHAR  ShiftCount
  );
```

RtlLargeIntegerShiftLeft is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerShiftRight

```
LARGE_INTEGER
  RtlLargeIntegerShiftRight(
  IN LARGE_INTEGER  LargeInteger,
  IN CCHAR  ShiftCount
  );
```

RtlLargeIntegerShiftRight is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLargeIntegerSubtract

```
LARGE_INTEGER
  RtlLargeIntegerSubtract(
  IN LARGE_INTEGER  Minuend,
  IN LARGE_INTEGER  Subtrahend
  );
```

RtlLargeIntegerSubtract is exported to support existing driver binaries and is obsolete. For better performance, use the compiler support for 64-bit integer operations.

RtlLengthSecurityDescriptor

```
ULONG
  RtlLengthSecurityDescriptor(
  IN PSECURITY_DESCRIPTOR  SecurityDescriptor
  );
```

RtlLLengthSecurityDescriptor returns the size of a given security descriptor.

Parameters

SecurityDescriptor

Points to a security descriptor.

Include

ntddk.h

Return Value

RtlLengthSecurityDescriptor returns the size in bytes of the descriptor.

Comments

Callers of **RtlLengthSecurityDescriptor** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlCreateSecurityDescriptor, **RtlSetDaclSecurityDescriptor**, **RtlValidSecurity-Descriptor**

RtlMoveMemory

```
VOID
  RtlMoveMemory(
  IN VOID UNALIGNED  *Destination,
  IN CONST VOID UNALIGNED  *Source,
  IN SIZE_T  Length
  );
```

RtlMoveMemory moves memory either forward or backward, aligned or unaligned, in 4-byte blocks, followed by any remaining bytes.

Parameters

Destination

Points to the destination of the move.

Source

Points to the memory to be copied.

Length

Specifies the number of bytes to be copied.

Include

wdm.h or *ntddk.h*

Comments

The (*Source* + *Length*) can overlap the *Destination* range passed in to **RtlMoveMemory**.

Callers of **RtlMoveMemory** can be running at any IRQL if both memory blocks are resident. Otherwise, the caller must be running at IRQL < DISPATCH_LEVEL.

See Also

RtlCopyMemory

RtlNumberOfClearBits

```
ULONG
  RtlNumberOfClearBits(
  IN PRTL_BITMAP  BitMapHeader
  );
```

RtlNumberOfClearBits returns a count of the clear bits in a given bitmap variable.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

Include

ntddk.h

Return Value

RtlNumberOfClearBits returns how many bits currently are clear.

Comments

Callers of **RtlNumberOfClearBits** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlNumberOfClearBits** can be called at any IRQL.

See Also

RtlFindClearBits, **RtlFindClearRuns**, **RtlFindFirstRunClear**, **RtlFindLastBackwardRunClear**, **RtlFindLongestRunClear**, **RtlFindNextForwardRunClear**, **RtlInitializeBitMap**, **RtlNumberOfSetBits**

RtlNumberOfSetBits

```
ULONG
  RtlNumberOfSetBits(
  IN PRTL_BITMAP  BitMapHeader
  );
```

RtlNumberOfSetBits returns a count of the set bits in a given bitmap variable.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

Include

ntddk.h

Return Value

RtlNumberOfSetBits returns how many bits currently are set.

Comments

Callers of **RtlNumberOfSetBits** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlNumberOfSetBits** can be called at any IRQL.

See Also

RtlFindSetBits, **RtlInitializeBitMap**, **RtlNumberOfClearBits**

RtlPrefixUnicodeString

```
BOOLEAN
  RtlPrefixUnicodeString(
  IN PUNICODE_STRING String1,
  IN PUNICODE_STRING String2,
  IN BOOLEAN CaseInSensitive
  );
```

RtlPrefixUnicodeString compares two Unicode strings to determine whether one string is a prefix of the other.

Parameters

String1

Points to the first string, which might be a prefix of the buffered Unicode string at *String2*.

String2

Points to the second string.

CaseInSensitive

If TRUE, case should be ignored when doing the comparison.

Include

ntddk.h

Return Value

RtlPrefixUnicodeString returns TRUE if *String1* is a prefix of *String2*.

Comments

Callers of **RtlPrefixUnicodeString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlCompareUnicodeString

RtlQueryRegistryValues

```
NTSTATUS
  RtlQueryRegistryValues(
  IN ULONG  RelativeTo,
  IN PCWSTR  Path,
  IN PRTL_QUERY_REGISTRY_TABLE  QueryTable,
  IN PVOID  Context,
  IN PVOID  Environment  OPTIONAL
  );
```

RtlQueryRegistryValues allows the caller to query several values from the registry subtree with a single call.

Parameters

RelativeTo

Specifies whether *Path* is an absolute registry path or is relative to a predefined path as one of the following:

Value	Meaning
RTL_REGISTRY_ABSOLUTE	Path is an absolute registry path.
RTL_REGISTRY_SERVICES	Path is relative to **\Registry\Machine\System\CurrentControlSet\Services**.
RTL_REGISTRY_CONTROL	Path is relative to **\Registry\Machine\System\CurrentControlSet\Control**.
RTL_REGISTRY_WINDOWS_NT	Path is relative to **\Registry\Machine\Software\Microsoft\Windows NT\CurrentVersion**.
RTL_REGISTRY_DEVICEMAP	Path is relative to **\Registry\Machine\Hardware\DeviceMap**.
RTL_REGISTRY_USER	Path is relative to **\Registry\User\CurrentUser**.
RTL_REGISTRY_OPTIONAL	Specifies that the key referenced by this parameter and the *Path* parameter are optional.
RTL_REGISTRY_HANDLE	Specifies that the *Path* parameter is actually a registry handle to use. This value is optional.

Path

Points to either an absolute registry path or a path relative to the known location specified by the *RelativeTo* parameter. Note that the names of keys in such a path must be known to the caller, including the last key in the path. If the RTL_REGISTRY_HANDLE flag is specified, this parameter is a registry handle for an already opened key to be queried directly.

QueryTable

Points to a table of one or more value names and subkey names in which the caller is interested. Each table entry contains a caller-supplied **QueryRoutine** that will be called for each value name that exists in the registry. The table must be terminated with a NULL table entry, which is a table entry with a NULL **QueryRoutine** and a NULL **Name** field. The *QueryTable* structure is defined as follows:

```
typedef struct _RTL_QUERY_REGISTRY_TABLE {
    PRTL_QUERY_REGISTRY_ROUTINE QueryRoutine;
    ULONG Flags;
    PWSTR Name;
    PVOID EntryContext;
    ULONG DefaultType;
    PVOID DefaultData;
    ULONG DefaultLength;
} RTL_QUERY_REGISTRY_TABLE, *PRTL_QUERY_REGISTRY_TABLE;
```

This routine is called with the name, type, data, and data length of a registry value. If this field is NULL, it marks the end of the table.

A caller-supplied **QueryRoutine** is declared as follows:

```
NTSTATUS
(*PRTL_QUERY_REGISTRY_ROUTINE)(
    IN PWSTR ValueName,
    IN ULONG ValueType,
    IN PVOID ValueData,
    IN ULONG ValueLength,
    IN PVOID Context,
    IN PVOID EntryContext
    );
```

The remaining members of the *QueryTable* structure include those on the following pages.

Flags

Control how the remaining fields are to be interpreted, as follows:

Value	Meaning
RTL_QUERY_REGISTRY_SUBKEY	The **Name** of this table entry is another path to a registry key, and all following table entries are for that key rather than the key specified by the *Path* parameter. This change in focus lasts until the end of the table or until another RTL_REGISTRY_SUBKEY or RTL_QUERY_REGISTRY_TOPKEY entry is seen. Each such entry must specify a path that is relative to the *Path* specified in the call to **RtlQueryRegistryValues**.
RTL_QUERY_REGISTRY_TOPKEY	Resets the current registry key handle to the original one specified by the *RelativeTo* and *Path* parameters. This is useful for getting back to the original node after descending into subkeys with the RTL_QUERY_REGISTRY SUBKEY flag.
RTL_QUERY_REGISTRY_REQUIRED	Specifies that this value is required and, if it is not found, STATUS_OBJECT_NAME_NOT_FOUND is returned. This is used for a table entry that specifies a NULL **Name** so that **RtlQueryRegistryValues** will enumerate all of the value names under a key and return STATUS_OBJECT_NAME_NOT_FOUND only if there are no value keys under the current key.
RTL_QUERY_REGISTRY_NOVALUE	Specifies that even though there is no **Name** for this table entry, all the caller wants is a callback: that is, the caller does *not* want to enumerate all the values under the current key. The **QueryRoutine** is called with NULL for *ValueData*, REG_NONE for *ValueType*, and zero for *ValueLength*.
RTL_QUERY_REGISTRY_NOEXPAND	Specifies that, if the type of this registry value is REG_EXPAND_SZ or REG_MULTI_SZ, **RtlQueryRegistryValues** is *not* to do any preprocessing of these registry values before calling the **QueryRoutine**. By default, **RtlQueryRegistryValues** expands environment variable references into REG_EXPAND_SZ values, enumerates each zero-terminated string in a REG_MULTI_SZ value, and calls the **QueryRoutine** once with each, making it look like there is more than one REG_SZ value with the same *ValueName*.

Value	Meaning
RTL_QUERY_REGISTRY_DIRECT	The **QueryRoutine** is ignored, and the **EntryContext** points to the location to store the value. For zero-terminated strings, **EntryContext** points to a UNICODE_STRING structure that describes the maximum size of the buffer. If the **Buffer** is NULL, a buffer is allocated.
RTL_QUERY_REGISTRY_DELETE	This is used to delete value keys after they have been queried.

This is the name of a Value that the caller queried. If **Name** is NULL, the **QueryRoutine** specified for this table entry is called for all values associated with the current registry key.

EntryContext

This is an arbitrary 32-bit field that is passed uninterpreted each time the **QueryRoutine** is called.

DefaultType

Specifies the REG_*XXX* type of the data to be queried if **Flags** is not set with RTL_REGISTRY_REQUIRED; otherwise, zero.

DefaultData

Points to default value(s) for a named value entry of the **DefaultType** if it is not already in the registry under **Name** and if **Flags** is not set with RTL_REGISTRY_REQUIRED; otherwise, NULL.

DefaultLength

If there is no value name that matches the name given by the **Name**, and the **DefaultType** field is not REG_NONE, the **QueryRoutine** for this table entry is called with the contents of the following fields as if the value had been found in the registry. If the **DefaultType** is REG_SZ, REG_EXPANDSZ, or REG_MULTI_SZ and the **DefaultLength** is zero, the value of **DefaultLength** will be computed based on the length of the Unicode string pointed to by **DefaultData**.

Context

Points to a context that is passed, uninterpreted, when each **QueryRoutine** is called.

Environment

Points to the environment used when expanding variable values in REG_EXPAND_SZ registry values, or a NULL pointer (optional).

Include

wdm.h or *ntddk.h*

Return Value

RtlQueryRegistryValues returns STATUS_SUCCESS if the entire *QueryTable* was processed successfully. If an error occurs, **RtlQueryRegistryValues** returns an error status such as:

STATUS_INVALID_PARAMETER
STATUS_OBJECT_NAME_NOT_FOUND
STATUS_*XXX* (an error status from a user-supplied **QueryRoutine**)

Comments

The caller specifies an initial key path and a table. The table contains one or more entries that describe the key values and subkey names in which the caller is interested. **RtlQueryRegistryValues** starts at the initial key and enumerates the entries in the table.

For each entry specifying a value name or subkey name that exists in the registry, **RtlQueryRegistryValues** calls the **QueryRoutine** associated with each table entry. Each entry's caller-supplied **QueryRoutine** is passed the value name, type, data, and data length.

When building the *QueryTable*, be sure to allocate an entry for each value being queried, plus a NULL entry at the end. Zero the table and then initialize the entries. *QueryTable* must be allocated from resident memory (nonpaged pool).

If an error occurs at any stage of processing the *QueryTable*, **RtlQueryRegistryValues** stops processing the table and returns the error status.

Callers of **RtlQueryRegistryValues** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlZeroMemory, **ZwEnumerateKey**, **ZwOpenKey**

RtlRetrieveUlong

```
VOID
  RtlRetrieveUlong(
  IN OUT PULONG  DestinationAddress,
  IN PULONG  SourceAddress
  );
```

RtlRetrieveUlong retrieves a ULONG value from the source address, avoiding alignment faults. The destination address is assumed to be aligned.

Parameters

DestinationAddress

Points to a ULONG-aligned location in which to store the ULONG value.

SourceAddress

Points to a location from which to retrieve the ULONG value.

Include

wdm.h or *ntddk.h*

Comments

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 < DISPATCH_LEVEL.

See Also

RtlStoreUlong

RtlRetrieveUshort

```
VOID
  RtlRetrieveUshort(
  IN OUT PUSHORT  DestinationAddress,
  IN PUSHORT  SourceAddress
  );
```

RtlRetrieveUshort retrieves a USHORT value from the source address, avoiding alignment faults.

Parameters

DestinationAddress

Points to a USHORT-aligned location in which to store the value.

SourceAddress

Points to a location from which to retrieve the value.

Include

wdm.h or *ntddk.h*

Comments

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 < DISPATCH_LEVEL.

See Also

RtlStoreUshort

RtlSetAllBits

```
VOID
  RtlSetAllBits(

 IN PRTL_BITMAP  BitMapHeader
  );
```

RtlSetAllBits sets all bits in a given bitmap variable.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

Include

ntddk.h

Comments

Callers of **RtlSetAllBits** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlSetAllBits** can be called at any IRQL.

See Also

RtlAreBitsClear, **RtlAreBitsSet**, **RtlFindClearBits**, **RtlFindClearBitsAndSet**, **RtlFindClearRuns**, **RtlFindFirstRunClear**, **RtlInitializeBitMap**, **RtlFindLastBackwardRunClear**, **RtlFindLongestRunClear**, **RtlFindNextForwardRunClear**, **RtlInitializeBitMap**, **RtlSetBits**, **RtlNumberOfSetBits**

RtlSetBits

```
VOID
  RtlSetBits(
  IN PRTL_BITMAP  BitMapHeader,
  IN ULONG  StartingIndex,
  IN ULONG  NumberToSet
  );
```

RtlSetBits sets all bits in a given range of a given bitmap variable.

Parameters

BitMapHeader

Points to an initialized bitmap header for the caller's bitmap variable.

StartingIndex

Specifies the start of the bit range to be set. This is a zero-based value indicating the position of the first bit in the range.

NumberToSet

Specifies how many bits to set.

Include

ntddk.h

Comments

RtlSetBits simply returns control if the input *NumberToSet* is zero. *StartingIndex* plus *NumberToSet* must be less than or equal to **sizeof(***BitMapHeader***->SizeOfBitMap)**.

Callers of **RtlSetBits** must be running at IRQL < DISPATCH_LEVEL if the memory containing the bitmap variable or at *BitMapHeader* is pageable. Otherwise, **RtlSetBits** can be called at any IRQL.

See Also

RtlAreBitsClear, RtlFindClearBitsAndSet, RtlFindClearRuns, RtlFindFirstRunClear, RtlFindLastBackwardRunClear, RtlFindLongestRunClear, RtlFindNextForwardRunClear, RtlInitializeBitMap, RtlSetAllBits, RtlNumberOfClearBits

RtlSetDaclSecurityDescriptor

```
NTSTATUS
  RtlSetDaclSecurityDescriptor(
  IN OUT PSECURITY_DESCRIPTOR  SecurityDescriptor,
  IN BOOLEAN  DaclPresent,
  IN PACL  Dacl  OPTIONAL,
  IN BOOLEAN  DaclDefaulted  OPTIONAL
  );
```

RtlSetDaclSecurityDescriptor sets the DACL information of an absolute-format security descriptor. If there is already a DACL present in the security descriptor, it is superseded.

Parameters

SecurityDescriptor

Points to the security descriptor to which the DACL is to be applied.

DaclPresent

If FALSE, indicates that the *DaclPresent* flag in the security descriptor should be set to FALSE. In this case, the remaining optional parameters are ignored. Otherwise, the *DaclPresent* control flag in the security descriptor is set to TRUE and the remaining optional parameters are not ignored.

Dacl

Points to the DACL for the security descriptor. If this parameter is NULL, a NULL ACL is assigned to the security descriptor. A NULL DACL unconditionally grants access. The DACL is referenced by, but not copied into, the security descriptor.

DaclDefaulted

When set, indicates that the DACL was picked up from some default mechanism rather than explicitly specified by the caller. This value is set in the *DaclDefaulted* control flag in the security descriptor. If this parameter is NULL, the *DaclDefaulted* flag will be cleared.

Include

ntddk.h

Return Value

RtlSetDaclSecurityDescriptor can return one of the following:

STATUS_SUCCESS

The call completed successfully.

STATUS_UNKNOWN_REVISION

The revision of the security descriptor is unknown.

STATUS_INVALID_SECURITY_DESCR

The security descriptor is not an absolute format security descriptor.

Comments

Callers of **RtlSetDaclSecurityDescriptor** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlCreateSecurityDescriptor, **RtlLengthSecurityDescriptor**, **RtlValidSecurity-Descriptor**

RtlStoreUlong

```
VOID
  RtlStoreUlong(
  IN PULONG  Address,
  IN ULONG  Value
  );
```

RtlStoreUlong stores a ULONG value at a particular address, avoiding alignment faults.

Parameters

Address

Points to a location in which to store a given ULONG value.

Value

Specifies a ULONG value to be stored.

Include

wdm.h or *ntddk.h*

Comments

The caller can be running at any IRQL if *Address* points to nonpaged pool. Otherwise, the caller must be running at IRQL < DISPATCH_LEVEL.

See Also

RtlRetrieveUlong

RtlStoreUlonglong

```
VOID
RtlStoreUlonglong (
  PULONGLONG Address,
  ULONGLONG Value
  );
```

RtlStoreUlonglong stores a specified ULONGLONG value at a specified memory address, avoiding memory alignment faults.

Parameters

Address

Pointer to a memory location where the value of *Value* is stored.

Value

Specifies the ULONGLONG value that is stored.

Include

ntddk.h or *wdm.h*

Comments

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 < DISPATCH_LEVEL.

See Also

RtlStoreUlong, RtlStoreUlongPtr

RtlStoreUlongPtr

```
VOID
RtlStoreUlongPtr (
  PULONG_PTR Address,
  ULONG_PTR Value
  );
```

RtlStoreUlongPtr stores a specified ULONG_PTR value at a specified memory location, avoiding memory alignment faults.

Parameters

Address

Pointer to a memory location where the value of *Value* is stored.

Value

Specifies the ULONG_PTR value that is stored.

Include

ntddk.h or *wdm.h*

Comments

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 < DISPATCH_LEVEL.

See Also

RtlStoreUlong, **RtlStoreUlonglong**

RtlStoreUshort

```
VOID
  RtlStoreUshort(
  IN PUSHORT  Address,
  IN USHORT  Value
  );
```

RtlStoreUshort stores a USHORT value at a particular address, avoiding alignment faults.

Parameters

Address

Points to a location in which to store a USHORT value.

Value

Specifies a USHORT value to be stored.

Include

wdm.h or *ntddk.h*

Comments

The caller can be running at any IRQL if *Address* points to nonpaged pool. Otherwise, the caller must be running at IRQL < DISPATCH_LEVEL.

See Also

RtlRetrieveUshort

RtlStringFromGUID

```
NTSTATUS
  RtlStringFromGUID(
  IN REFGUID Guid,
  OUT PUNICODE_STRING GuidString
  );
```

RtlStringFromGUID converts a given GUID from binary format into a Unicode string.

Parameters

Guid

Specifies the binary-format GUID to convert.

GuidString

Points to a caller-supplied variable in which a pointer to the converted GUID string is returned. **RtlStringFromGUID** allocates the buffer space for the string, which the caller must free by calling **RtlFreeUnicodeString**.

Include

wdm.h or *ntddk.h*

Return Value

If the conversion succeeds, **RtlStringFromGUID** returns STATUS_SUCCESS. Otherwise, no storage was allocated, nor was the conversion performed.

Comments

Callers of **RtlStringFromGUID** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlGUIDFromString

RtlTimeFieldsToTime

```
BOOLEAN
  RtlTimeFieldsToTime(
  IN PTIME_FIELDS  TimeFields,
  IN PLARGE_INTEGER  Time
  );
```

RtlTimeFieldsToTime converts TIME_FIELDS information to a system time value.

Parameters

TimeFields

Points to the following structure, containing the time information to be converted:

```
typedef struct TIME_FIELDS {
    CSHORT Year;
    CSHORT Month;
    CSHORT Day;
    CSHORT Hour;
    CSHORT Minute;
    CSHORT Second;
    CSHORT Milliseconds;
    CSHORT Weekday;
} TIME_FIELDS;
```

Members

Year

Is in the range from 1601 on.

Month

Is in the range from 1 to 12.

Day

Is in the range from 1 to 31.

Hour

Is in the range from 0 to 23.

Minute

Is in the range from 0 to 59.

Second

Is in the range from 0 to 59.

Milliseconds

Is in the range from 0 to 999.

Weekday

Is in the range from 0 to 6 (Sunday to Saturday).

Time

Points to a buffer, which is to contain the converted system time value as a large integer.

Include

wdm.h or *ntddk.h*

Return Value

RtlTimeFieldsToTime returns TRUE if the input *TimeFields* data was successfully converted.

Comments

RtlTimeFieldsToTime ignores the **Weekday** value in *TimeFields*.

Callers of **RtlTimeFieldsToTime** can be running at any IRQL if both input buffers are resident.

See Also

ExLocalTimeToSystemTime, **ExSystemTimeToLocalTime**, **KeQuerySystemTime**, **RtlTimeToTimeFields**

RtlTimeToTimeFields

```
VOID
  RtlTimeToTimeFields(
  IN PLARGE_INTEGER  Time,
  IN PTIME_FIELDS  TimeFields
  );
```

RtlTimeToTimeFields converts system time into a TIME_FIELDS structure.

Parameters

Time

Points to a buffer containing the absolute system time as a large integer, accurate to 100-nanosecond resolution.

TimeFields

Points to a caller-allocated buffer, which must be at least **sizeof**(TIME_FIELDS), to contain the returned information.

Include

wdm.h or *ntddk.h*

Comments

Callers of **RtlTimeToTimeFields** can be running at any IRQL if both input buffers are resident.

See Also

ExLocalTimeToSystemTime, **ExSystemTimeToLocalTime**, **KeQuerySystemTime**, **RtlTimeFieldsToTime**

RtlUlongByteSwap

```
ULONG
FASTCALL
  RtlUlongByteSwap(
  IN ULONG Source
  );
```

RtlUlongByteSwap converts a ULONG from little-endian to big-endian, and vice versa.

Parameters

Source

ULONG to convert.

Include

wdm.h or ntddk.h

Return Value

The converted ULONG value.

See Also

RtlUlonglongByteSwap, **RtlUshortByteSwap**

RtlUlonglongByteSwap

```
ULONGLONG
FASTCALL
  RtlUlonglongByteSwap(
  IN ULONGLONG Source
  );
```

RtlUlonglongByteSwap converts a ULONGLONG from little-endian to big-endian, and vice versa.

Parameters

Source

ULONGLONG to convert.

Include

wdm.h or *ntddk.h*

Return Value

The converted ULONGLONG value.

See Also

RtlUlongByteSwap, **RtlUshortByteSwap**

RtlUnicodeStringToAnsiSize

```
ULONG
RtlUnicodeStringToAnsiSize(
  PUNICODE_STRING UnicodeString
  );
```

RtlUnicodeStringToAnsiSize returns the number of bytes required for a NULL-terminated ANSI string that is equivalent to a specified Unicode string.

Parameters

UnicodeString

Pointer to the Unicode string for which to compute the number of bytes required for an equivalent NULL-terminated ANSI string.

Include

wdm.h

Return Value

If the Unicode string can be translated into an ANSI string using the current system locale information, **RtlUnicodeStringToAnsiSize** returns the number of bytes required for an equivalent NULL-terminated ANSI string. Otherwise, **RtlUnicodeStringToAnsiSize** returns zero.

Comments

The Unicode string is interpreted for the current system locale.

RtlUnicodeStringToAnsiSize performs the same operation as **RtlxUnicodeStringToAnsiSize**, but executes faster if the system does not use multibyte code pages.

RtlUnicodeStringToAnsiSize runs at IRQL PASSIVE_LEVEL.

See Also

RtlxUnicodeStringToAnsiSize

RtlUnicodeStringToAnsiString

```
NTSTATUS
  RtlUnicodeStringToAnsiString(
  IN OUT PANSI_STRING  DestinationString  OPTIONAL,
  IN PUNICODE_STRING  SourceString,
  IN BOOLEAN  AllocateDestinationString
  );
```

RtlUnicodeStringToAnsiString converts a given Unicode string into an ANSI string. The translation is done in accord with the current system-locale information.

Parameters

DestinationString

Points to a caller-allocated buffer for the ANSI string or is NULL if *AllocateDestinationString* is set to TRUE. If the translation cannot be done because a character in the Unicode string does not map to an ANSI character in the current system locale, an error is returned.

SourceString

Points to the Unicode source string to be converted to ANSI.

AllocateDestinationString

TRUE if this routine is to allocate the buffer space for the *DestinationString*. If it does, the buffer must be deallocated by calling **RtlFreeAnsiString**.

Include

wdm.h or *ntddk.h*

Return Value

If the conversion succeeds, **RtlUnicodeStringToAnsiString** returns STATUS_SUCCESS. Otherwise, no storage was allocated, and no conversion was done.

Comments

Callers of **RtUnicodeStringToAnsiString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlAnsiStringToUnicodeString, RtlFreeAnsiString

RtlUnicodeStringToInteger

```
NTSTATUS
  RtlUnicodeStringToInteger(
  IN PUNICODE_STRING  String,
  IN ULONG  Base  OPTIONAL,
  OUT PULONG  Value
  );
```

RtlUnicodeStringToInteger converts a Unicode string representation of an integer into its integer equivalent.

Parameters

String

Points to the Unicode string to be converted to its integer equivalent.

Base

An optional argument that indicates the base of the number expressed as a Unicode string.

Value

Points to caller supplied storage of type ULONG. **RtlUnicodeStringToInteger** returns the integer conversion results in *Value*.

Include

wdm.h or *ntddk.h*

Return Value

If the conversion is successful, **RtlUnicodeStringToInteger** returns STATUS_SUCCESS and *Value* is set to the integer equivalent of the Unicode string. Otherwise, the *Value* is set to 0, and **RtlUnicodeStringToInteger** returns STATUS_INVALID_PARAMETER.

Comments

If the first character of the string is a "–", the sign of the output *Value* is negative, otherwise if the first character is a "+" or there is no sign character, the sign of *Value* is positive.

If no *Base* is supplied, **RtlUnicodeStringToInteger** checks for a leading character to indicate the base of the number. An "x" indicates the string is to be converted as a hexadecimal integer; an "o" indicates the string is to be converted as an octal integer; a "b" indicates the string is to be converted as a binary integer. Otherwise, **RtlUnicodeStringToInteger** assumes the number is to be converted as a base 10 integer.

Callers of **RtlUnicodeStringToInteger** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlIntegerToUnicodeString

RtlUpcaseUnicodeChar

```
WCHAR
  RtlUpcaseUnicodeChar(
  IN WCHAR  SourceCharacter
  );
```

RtlUpcaseUnicodeChar converts the specified Unicode character to uppercase.

Parameters

SourceCharacter

Specifies the character to convert.

Include

ntddk.h

Return Value

RtlUpcaseUnicodeChar returns the uppercase version of the specified Unicode character.

Comments

Callers of **RtlUpcaseUnicodeChar** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlUpcaseUnicodeString, RtlUpperChar

RtlUpcaseUnicodeString

```
NTSTATUS
  RtlUpcaseUnicodeString(
  IN OUT PUNICODE_STRING  DestinationString  OPTIONAL,
  IN PCUNICODE_STRING  SourceString,
  IN BOOLEAN  AllocateDestinationString
  );
```

RtlUpcaseUnicodeString converts a copy of the source string to upper case and writes the converted string in the destination buffer.

Parameters

DestinationString

Points to a caller-allocated buffer for the converted Unicode string or is NULL if *AllocateDestinationString* is set to TRUE.

SourceString

Points to the source Unicode string to be converted to upper case.

AllocateDestinationString

TRUE if **RtlUpcaseUnicodeString** is to allocate the buffer space for the *DestinationString*. If it does, the buffer must be deallocated by calling **RtlFreeUnicodeString**.

Include

ntddk.h

Return Value

If the operation succeeds, **RtlUpcaseUnicodeString** returns STATUS_SUCCESS. Otherwise, no storage was allocated, and no conversion was done.

Comments

Callers of **RtlUpcaseUnicodeString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlFreeUnicodeString, RtlUpcaseUnicodeChar

RtlUpperChar

```
CHAR
  RtlUpperChar(
  IN CHAR Character
  );
```

RtlUpperChar converts the specified character to uppercase.

Parameters

Character

Specifies the character to convert.

Include

ntddk.h

Return Value

RtlUpperChar returns the uppercase version of the specified character or returns the value specified by the caller for *Character* if the specified character cannot be converted.

Comments

RtlUpperChar returns the input *Character* unconverted if it is the lead byte of a multibyte character or if the uppercase equivalent of *Character* is a double-byte character. To convert such characters, use **RtlUpcaseUnicodeChar**.

Callers of RtlUpperChar must be running at IRQL PASSIVE_LEVEL.

See Also

RtlUpcaseUnicodeChar, **RtlUpperString**

RtlUpperString

```
VOID
  RtlUpperString(
  IN OUT PSTRING  DestinationString,
  IN PSTRING  SourceString
  );
```

RtlUpperString copies the given *SourceString* to the *DestinationString* buffer, converting it to uppercase.

Parameters

DestinationString

Points to the buffer for the converted destination string.

SourceString

Points to the source string to be converted to uppercase.

Include

ntddk.h

Comments

The **MaximumLength** and **Buffer** fields of *DestinationString* are not modified by this routine.

The number of bytes copied from *SourceString* is either the **Length** of *SourceString* or the **MaximumLength** of *DestinationString*, whichever is smaller.

Callers of **RtlUpperString** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlUpperChar

RtlUshortByteSwap

```
USHORT
  RtlUshortByteSwap(
  IN USHORT Source
  );
```

RtlUshortByteSwap converts a USHORT from little-endian to big-endian, and vice versa.

Parameters

Source

USHORT to convert.

Include

wdm.h or *ntddk.h*

Return Value

The converted USHORT value.

See Also

RtlUlongByteSwap, **RtlUlonglongByteSwap**

RtlValidSecurityDescriptor

```
BOOLEAN
  RtlValidSecurityDescriptor(
  IN PSECURITY_DESCRIPTOR  SecurityDescriptor
  );
```

RtlValidSecurityDescriptor checks a given security descriptor's validity.

Parameters

SecurityDescriptor

Points to the security descriptor to be checked.

Include

ntddk.h

Return Value

RtlValidSecurityDescriptor returns TRUE if the given descriptor is valid.

Comments

Callers of **RtlValidSecurityDescriptor** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlCreateSecurityDescriptor, **RtlLengthSecurityDescriptor**, **RtlSetDaclSecurity-Descriptor**

RtlVerifyVersionInfo

```
NTSTATUS
RtlVerifyVersionInfo(
  IN PRTL_OSVERSIONINFOEXW VersionInfo,
  IN ULONG TypeMask,
  IN ULONGLONG  ConditionMask
  );
```

RtlVerifyVersionInfo compares a specified set of operating system version requirements to the corresponding attributes of the currently running version of the operating system.

Parameters

VersionInfo

Pointer to an RTL_OSVERSIONINFOEXW structure that specifies the operating system version requirements to compare to the corresponding attributes of the currently running version of the operating system.

TypeMask

Specifies which members of *VersionInfo* to compare with the corresponding attributes of the currently running version of the operating system. *TypeMask* is set to a logical OR of one or more of the following values:

Value	Corresponding Member
VER_BUILDNUMBER	**dwBuildNumber**
VER_MAJORVERSION	**dwMajorVersion**
VER_MINORVERSION	**dwMinorVersion**
VER_PLATFORMID	**dwPlatformId**
VER_SERVICEPACKMAJOR	**wServicePackMajor**
VER_SERVICEPACKMINOR	**wServicePackMinor**
VER_SUITENAME	**wSuiteMask**
VER_PRODUCT_TYPE	**wProductType**

ConditionMask

Specifies how to compare each *VersionInfo* member. To set the value of *ConditionMask*, a caller should use the **VER_SET_CONDITION** macro:

```
VER_SET_CONDITION (
        IN OUT  ULONGLONG  ConditionMask,
        IN  ULONG   TypeBitMask,
        IN  UCHAR   ComparisonType
        );
```

The value of *ConditionMask* is created in the following way:

- Initialize the value of *ConditionMask* to zero.
- Call **VERSION_SET_CONDITION** once for each *VersionInfo* member specified by *TypeMask*.
- Set the *TypeBitMask* and *ComparisonType* parameters for each call to **VERSION_SET_CONDITION** as follows:

TypeBitMask

Indicates the *VersionInfo* member for which the comparison type is set. *TypeBitMask* can be one of the following values:

Value	Corresponding Member
VER_BUILDNUMBER	**dwBuildNumber**
VER_MAJORVERSION	**dwMajorVersion**
VER_MINORVERSION	**dwMinorVersion**
VER_PLATFORMID	**dwPlatformId**
VER_SERVICEPACKMAJOR	**wServicePackMajor**
VER_SERVICEPACKMINOR	**wServicePackMinor**
VER_SUITENAME	**wSuiteMask**
VER_PRODUCT_TYPE	**wProductType**

ComparisonType

Specifies the comparison type that **RtlVerifyVersionInfo** uses to compare the *VersionInfo* member specified by *TypeBitMask* with the corresponding attribute of the currently running operating system.

For all values of *TypeBitMask* other than VER_SUITENAME, *ComparsionType* is set to one of the following values:

Value	Meaning
VER_EQUAL	The current value must be equal to the specified value.
VER_GREATER	The current value must be greater than the specified value.
VER_GREATER_EQUAL	The current value must be greater than or equal to the specified value.
VER_LESS	The current value must be less than the specified value.
VER_LESS_EQUAL	The current value must be less than or equal to the specified value.

If *TypeBitMask* is set to VER_SUITENAME, *ComparisonType* is set to of one the following values:

Value	Meaning
VER_AND	All product suites specified in the **wSuiteMask** member must be present in the current system.
VER_OR	At least one of the specified product suites must be present in the current system.

Include

ntddk.h

Return Value

RtlVerifyVersionInfo returns one of the following status values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

The input parameters are not valid.

STATUS_REVISION_MISMATCH

The specified version does not match the currently running version of the operating system.

Comments

RtlVerifyVersionInfo enables a driver to easily verify the presence of a required set of operating system attributes. **RtlVerifyVersionInfo** is the kernel-mode equivalent of the user-mode **VerifyVersionInfo** function in the Platform SDK. See the example in the Platform SDK that shows how to verify the system version.

Typically, **RtlVerifyVersionInfo** returns STATUS_SUCCESS only if all comparisons succeed. However, the major version, minor version, and service pack version are tested in a sequential manner in the following way:

- If the major version exceeds the minimum required, then the minor version and service pack version are not tested. For example, if the current major version is 6.0, a test for a system greater than or equal to version 5.1 service pack 1 succeeds. The minor version and service pack version are not tested.
- If the minor version exceeds the minimum required, then the service pack version is not tested. For example, if the current major version is 5.2, a test for a system version greater than or equal to version 5.1 service pack 1 succeeds. The service pack version is not tested.
- If the major service pack version exceeds the minimum required, then the minor service pack version is not tested.

To verify a range of system versions, a driver can call **RtlVerifyVersionInfo** twice, once to verify a lower-bound on the system version and once to verify an upper-bound on the system version.

See Also

RtlGetVersion, RTL_OSVERSIONINFOW, RTL_OSVERSIONINFOEXW

RtlVolumeDeviceToDosName

```
NTSTATUS
RtlVolumeDeviceToDosName(
    IN  PVOID           VolumeDeviceObject,
    OUT PUNICODE_STRING DosName
    );
```

RtlVolumeDeviceToDosName returns the MS-DOS® path for a specified device object that represents a file system volume.

Parameters

VolumeDeviceObject

Pointer to a device object that represents a file system volume.

DosName

Pointer to a Unicode string containing the MS-DOS path of the volume device object specified by *VolumeDeviceObject*.

Include

ntddk.h

Return Value

RtlVolumeDeviceToDosName returns STATUS_SUCCESS or an appropriate error status.

Comments

RtlVolumeDeviceToDosName allocates the Unicode string buffer for the MS-DOS path from the memory pool. After the buffer is no longer required, a caller of this routine should use **ExFreePool** to free it.

RtlWriteRegistryValue

```
NTSTATUS
  RtlWriteRegistryValue(
  IN ULONG  RelativeTo,
  IN PCWSTR  Path,
  IN PCWSTR  ValueName,
  IN ULONG  ValueType,
  IN PVOID  ValueData,
  IN ULONG  ValueLength
  );
```

RtlWriteRegistryValue writes caller-supplied data into the registry along the specified relative path at the given value name.

Parameters

RelativeTo

Specifies whether *Path* is an absolute registry path or is relative to a predefined path as one of the following:

Value	Meaning
RTL_REGISTRY_ABSOLUTE	Path is an absolute registry path.
RTL_REGISTRY_SERVICES	Path is relative to **\Registry\Machine\System\CurrentControlSet\Services**.
RTL_REGISTRY_CONTROL	Path is relative to **\Registry\Machine\System\CurrentControlSet\Control**.
RTL_REGISTRY_WINDOWS_NT	Path is relative to **\Registry\Machine\Software\Microsoft\Windows NT\CurrentVersion**.
RTL_REGISTRY_DEVICEMAP	Path is relative to **\Registry\Machine\Hardware\DeviceMap**.
RTL_REGISTRY_USER	Path is relative to **\Registry\User\CurrentUser**.
RTL_REGISTRY_OPTIONAL	Specifies that the key referenced by this parameter and the *Path* parameter are optional.
RTL_REGISTRY_HANDLE	Optional; specifies that the *Path* parameter is actually a registry handle to use.

Path

Points to either an absolute registry path or a path relative to the known location specified by the *RelativeTo* parameter. If the RTL_REGISTRY_HANDLE flag is specified, this parameter is a registry handle for an already opened key to be used directly.

ValueName

Points to the name of a subkey or value entry to be written into the registry.

ValueType

Points to the value type, identified by the *ValueName* parameter, to be placed in the registry.

ValueData

Points to the name of a subkey or values for its value entries (or both) to be written into the registry.

ValueLength

Specifies the number of bytes of *ValueData* to be written into the registry.

Include

wdm.h or *ntddk.h*

Return Value

RtlWriteRegistryValue returns the status of the operation, either STATUS_SUCCESS or an error status.

Comments

Callers of **RtlWriteRegistryValue** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlCheckRegistryKey, **RtlCreateRegistryKey**, **RtlDeleteRegistryValue**, **RtlQueryRegistryValues**, **ZwOpenKey**

RtlxUnicodeStringToAnsiSize

```
ULONG
RtlxUnicodeStringToAnsiSize(
  PUNICODE_STRING UnicodeString
  );
```

RtlxUnicodeStringToAnsiSize returns the number of bytes required for a NULL-terminated ANSI string that is equivalent to a specified Unicode string.

Parameters

UnicodeString

Pointer to the Unicode string for which to compute the number of bytes required for an equivalent NULL-terminated ANSI string.

Include

wdm.h

Return Value

If the Unicode string can be translated into an ANSI string using the current system locale information, **RtlxUnicodeStringToAnsiSize** returns the number of bytes required for an equivalent NULL-terminated ANSI string. Otherwise, it returns zero.

Comments

The Unicode string is interpreted for the current system locale.

RtlxUnicodeStringToAnsiSize runs at IRQL PASSIVE_LEVEL.

See Also

RtlUnicodeStringToAnsiSize

RtlZeroBytes

```
VOID
  RtlZeroBytes(
  PVOID Destination,
  SIZE_T Length
  );
```

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**.

RtlZeroMemory

```
VOID
  RtlZeroMemory(
  IN VOID UNALIGNED  *Destination,
  IN SIZE_T  Length
  );
```

RtlZeroMemory fills a block of memory with zeros, given a pointer to the block and the length, in bytes, to be filled.

Parameters

Destination

Points to the memory to be filled with zeros.

Length

Specifies the number of bytes to be zeroed.

Include

wdm.h or *ntddk.h*

Comments

Callers of **RtlZeroMemory** can be running at any IRQL if the *Destination* block is in nonpaged pool. Otherwise, the caller must be running at IRQL < DISPATCH_LEVEL.

See Also

RtlFillMemory

CHAPTER 10

Security Reference Monitor Routines

Generally, higher-level drivers, particularly network drivers, call these routines.

References for the **Se***Xxx* routines are in alphabetical order. For an overview of the functionality of these routines, see Chapter 1, *Summary of Kernel-Mode Support Routines.*

SeAccessCheck

```
BOOLEAN
  SeAccessCheck(
  IN PSECURITY_DESCRIPTOR  SecurityDescriptor,
  IN PSECURITY_SUBJECT_CONTEXT  SubjectSecurityContext,
  IN BOOLEAN  SubjectContextLocked,
  IN ACCESS_MASK  DesiredAccess,
  IN ACCESS_MASK  PreviouslyGrantedAccess,
  OUT PPRIVILEGE_SET  *Privileges  OPTIONAL,
  IN PGENERIC_MAPPING  GenericMapping,
  IN KPROCESSOR_MODE  AccessMode,
  OUT PACCESS_MASK  GrantedAccess,
  OUT PNTSTATUS  AccessStatus
  );
```

SeAccessCheck determines whether the requested access rights can be granted to an object protected by a security descriptor and an object owner.

Parameters

SecurityDescriptor

Points to the security descriptor protecting the object being accessed.

SubjectSecurityContext

Points to the subject's captured security context.

SubjectContextLocked

Indicates whether the user's subject context is locked, so that it does not have to be locked again.

DesiredAccess

Specifies the access mask for rights that the caller is attempting to acquire.

PreviouslyGrantedAccess

Specifies the access rights already granted, for example, as a result of holding a privilege.

Privileges

Points to a caller-supplied variable to be set to the address of buffered privileges that will used as part of the access validation, or this parameter can be NULL. Such a buffer, if any, must be released by the caller with **ExFreePool** when the caller has consumed this information.

GenericMapping

Points to the generic mapping associated with this object type.

AccessMode

Specifies the access mode to be used in the check, one of **UserMode** or **KernelMode**.

GrantedAccess

Points to a returned access mask indicating the granted access.

AccessStatus

Points to the status value indicating why access was denied.

Include

ntddk.h

Return Value

If access is allowed, **SeAccessCheck** returns TRUE.

Comments

Network transport drivers call this routine.

SeAccessCheck might perform privilege tests for **SeTakeOwnershipPrivilege** and/or **SeSecurityPrivilege**, depending on the accesses being requested. It might perform additional privilege testing in future releases of the operating system.

This routine also might check whether the subject is the owner of the object in order to grant WRITE_DAC access.

If this routine returns FALSE, the caller should use the returned *AccessStatus* as its return value. That is, the caller should avoid hardcoding a return value of STATUS_ACCESS_DENIED or any other specific STATUS_*XXX* value.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

ExFreePool, IoGetFileObjectGenericMapping, SeValidSecurityDescriptor

SeAssignSecurity

```
NTSTATUS
  SeAssignSecurity(
  IN PSECURITY_DESCRIPTOR  ParentDescriptor  OPTIONAL,
  IN PSECURITY_DESCRIPTOR  ExplicitDescriptor  OPTIONAL,
  OUT PSECURITY_DESCRIPTOR  *NewDescriptor,
  IN BOOLEAN  IsDirectoryObject,
  IN PSECURITY_SUBJECT_CONTEXT  SubjectContext,
  IN PGENERIC_MAPPING  GenericMapping,
  IN POOL_TYPE  PoolType
  );
```

SeAssignSecurity builds a self-relative security descriptor for a new object, given the security descriptor of its parent directory and any originally requested security for the object.

Parameters

ParentDescriptor

Points to a buffer containing the security descriptor of the parent directory, if any, containing the new object being created. *ParentDescriptor* can be NULL, or have a NULL system access control list (SACL) or a NULL discretionary access control list (DACL).

ExplicitDescriptor

Points to a buffer containing the security descriptor specified by the user that is applied to the new object. *ExplicitDescriptor* can be NULL, or have a NULL SACL or a NULL DACL.

NewDescriptor

Receives a pointer to the returned security descriptor for which this routine allocates a buffer according to the given *PoolType*. The buffer is allocated from the paged memory pool.

IsDirectoryObject

Specifies whether the new object is a directory object. TRUE indicates the object contains other objects.

SubjectContext

Points to a buffer containing the security context of the subject creating the object. This is used to retrieve default security information for the new object, such as the default owner, the primary group, and discretionary access control.

GenericMapping

Points to an array of access mask values denoting the mapping between each generic right to nongeneric rights.

PoolType

Specifies the pool type to use when allocating a new security descriptor. Currently, *PoolType* is not used.

Include

ntddk.h

Return Value

SeAssignSecurity can return one of the following:

STATUS_SUCCESS

The assignment was successful.

STATUS_INVALID_OWNER

The SID provided for the owner of the target security descriptor is not one the caller is authorized to assign as the owner of an object.

STATUS_PRIVILEGE_NOT_HELD

The caller does not have the privilege (**SeSecurityPrivilege**) necessary to explicitly assign the specified system ACL.

Comments

Network transport drivers call this routine.

The final security descriptor returned to the caller may contain a mix of information, some explicitly provided from the new object's parent.

SeAssignSecurity assumes privilege checking has not been performed. This routine performs privilege checking.

The assignment of system and discretionary ACLs is governed by the logic illustrated in the following table:

	Explicit (nondefault) ACL specified	Explicit default ACL specified	No ACL specified
Inheritable ACL from parent	Assign specified ACL	Assign inherited ACL	Assign inherited ACL
No inheritable ACL from parent	Assign specified ACL	Assign default ACL	Assign no ACL

An explicitly specified ACL, whether a default ACL or not, can be empty or null. The caller must be a kernel-mode client or be appropriately privileged to explicitly assign a default or nondefault system ACL.

The assignment of the new object's owner and group is governed by the following logic:

- If the passed security descriptor includes an owner, it is assigned as the new object's owner. Otherwise, the caller's token is considered to determine the owner. Within the token, the default owner, if any, is assigned. Otherwise, the caller's user ID is assigned.
- If the passed security descriptor includes a group, it is assigned as the new object's group. Otherwise, the caller's token is considered to determine the group. Within the token, the default group, if any, is assigned. Otherwise, the caller's primary group ID is assigned.

Callers of **SeAssignSecurity** must be running at IRQL PASSIVE_LEVEL.

See Also

IoGetFileObjectGenericMapping, **SeDeassignSecurity**

SeAssignSecurityEx

```
NTSTATUS
SeAssignSecurityEx (
    IN PSECURITY_DESCRIPTOR ParentDescriptor OPTIONAL,
    IN PSECURITY_DESCRIPTOR ExplicitDescriptor OPTIONAL,
    OUT PSECURITY_DESCRIPTOR *NewDescriptor,
    IN GUID *ObjectType OPTIONAL,
    IN BOOLEAN IsDirectoryObject,
    IN ULONG AutoInheritFlags,
    IN PSECURITY_SUBJECT_CONTEXT SubjectContext,
    IN PGENERIC_MAPPING GenericMapping,
    IN POOL_TYPE PoolType
    );
```

SeAssignSecurityEx builds a self-relative security descriptor for a new object given the following optional parameters: a security descriptor of the object's parent directory, an explicit security descriptor for the object, and the object type.

Parameters

ParentDescriptor

Pointer to a security descriptor of the parent object that contains the new object being created. *ParentDescriptor* can be NULL, or have a NULL system access control list (SACL) or a NULL discretionary access control list (DACL).

ExplicitDescriptor

Pointer to an explicit security descriptor that is applied to the new object. *ExplicitDescriptor* can be NULL, or have a NULL SACL or a NULL DACL.

NewDescriptor

Pointer to a new security descriptor. **SeAssignSecurityEx** allocates the buffer for the new security descriptor from the paged memory pool.

ObjectType

Pointer to a GUID for the type of object being created. If the object does not have a GUID, *ObjectType* must be set to NULL.

IsDirectoryObject

Specifies whether the new object is a directory object. If *IsDirectoryObject* is set to TRUE, the new object is a directory object, otherwise the new object is not a directory object.

AutoInheritFlags

Specifies the type of automatic inheritance that is applied to access control entries (ACE) in the access control lists (ACL) specified by *ParentDescriptor*. *AutoInheritFlags* also controls privilege checking, owner checking, and setting a default owner and group for *NewDescriptor*. *AutoInheritFlags* must be set to a logical OR of one or more of the following values:

Value	Meaning
SEF_DACL_AUTO_INHERIT	ACEs in the DACL of *ParentDescriptor* are inherited by *NewDescriptor*, in addition to explicit ACEs specified by *ExplicitDescriptor*.
SEF_SACL_AUTO_INHERIT	ACEs in the SACL of *ParentDescriptor* are inherited by *NewDescriptor*, in addition to explicit ACEs specified by *ExplicitDescriptor*.

Value	Meaning
SEF_DEFAULT_DESCRIPTOR_FOR_OBJECT	*ExplicitDescriptor* is the default descriptor for the object type specified by *ObjectType*. *ExplicitDescriptor* is not used if ACEs are inherited from *ParentDescriptor*.
SEF_AVOID_PRIVILEGE_CHECK	Privilege checking is not done. This flag is useful with automatic inheritance because it avoids privilege checking on each child that needs to be updated.
SEF_AVOID_OWNER_CHECK	Owner checking is not done.
SEF_DEFAULT_OWNER_FROM_PARENT	If an owner is specified by *ExplicitDescriptor*, this flag is not used, and the owner of *NewDescriptor* is set to the owner specified by *ExplictDescriptor*. If an owner is not specified by *ExplicitDescriptor*, this flag is used in the following way: If the flag is set, the owner of *NewDescriptor* is set to the owner of *ParentDescriptor*. Otherwise, the owner of *NewDescriptor* is set to the owner specified by the *SubjectContext*.
SEF_DEFAULT_GROUP_FROM_PARENT	If a group is specified by *ExplicitDescriptor*, this flag is not used, and the group of *NewDescriptor* is set to the group specified by *ExplictDescriptor*. If a group is not specified by *ExplicitDescriptor*, this flag is used in the following way: If the flag is set, the group of *NewDescriptor* is set to the group of *ParentDescriptor*. Otherwise, the group of *NewDescriptor* is set to the group specified by the *SubjectContext*.

The assignment of system and discretionary ACLs is described in the following table:

	Non-default explicit descriptor(1)	Default explicit descriptor(2)	NULL explicit descriptor
ACL is inherited from parent descriptor(3).	Assign both inherited and explicit ACLs(5)(6).	Assign inherited ACL.	Assign inherited ACL.
ACL is not inherited from parent descriptor(4).	Assign non-default ACL.	Assign default ACL.	Assign no ACL.

Assignment Notes

1. The SEF_DEFAULT_DESCRIPTOR_FOR_OBJECT flag is not specified.
2. The SEF_DEFAULT_DESCRIPTOR_FOR_OBJECT flag is specified.
3. The auto inherit flag for an ACL is specified (SEF_DACL_AUTO_INHERIT or SEF_SACL_AUTO_INHERIT).

4. The automatic inherit flag for an ACL is not specified.
5. ACEs with the INHERITED_ACE bit set in their **AceFlags** member are *not* copied to the assigned security descriptor.
6. ACEs that are inherited from the parent descriptor are appended after the ACEs specified by the explicit descriptor.

SubjectContext

Pointer to a security context of the subject that is creating the object. *SubjectContext* is used to retrieve default security information for the new object, including the default owner, the primary group, and discretionary access control.

GenericMapping

Pointer to an array of access mask values that specify the mapping between each generic rights to object-specific rights.

PoolType

Specifies the type of memory pool for *NewDescriptor*. Currently, *PoolType* is not used.

Include

ntddk.h

Return Value

SeAssignSecurityEx returns one of the following values:

STATUS_SUCCESS

The assignment was successful.

STATUS_INVALID_OWNER

The SID provided as the owner of the new security descriptor is not a SID that the caller is authorized to assign as the owner of an object.

STATUS_PRIVILEGE_NOT_HELD

The caller does not have the privilege (**SeSecurityPrivilege**) necessary to explicitly assign the specified SACL.

Comments

SeAssignSecurityEx extends the basic operation of **SeAssignSecurity** in the following ways:

- *ObjectType* optionally specifies an object type. Object-specific inheritance is controlled by the following members of an object-specific ACE: Flags, **InheritedObjectType**, and **Header.AceFlags**.
- *AutoInheritFlags* specifies the type of automatic inheritance of ACEs that is used. *AutoInheritFlags* also controls privilege checking, owner checking, and setting a default owner and group for *NewDescriptor*.

For more information on security and access control, see the documentation on these topics in the Platform SDK.

SeAssignSecurityEx runs at IRQL PASSIVE_LEVEL.

See Also

SeAssignSecurity, **SeDeassignSecurity**

SeDeassignSecurity

```
NTSTATUS
  SeDeassignSecurity(
  IN OUT PSECURITY_DESCRIPTOR  *SecurityDescriptor
  );
```

SeDeassignSecurity deallocates the memory associated with a security descriptor that was assigned using **SeAssignSecurity**.

Parameters

SecurityDescriptor

Points to the buffered security descriptor being released.

Include

ntddk.h

Return Value

If the deallocation succeeds, **SeDeassignSecurity** returns STATUS_SUCCESS.

Comments

Callers of **SeDeassignSecurity** must be running at IRQL PASSIVE_LEVEL.

See Also

SeAssignSecurity

SeSinglePrivilegeCheck

```
BOOLEAN
  SeSinglePrivilegeCheck(
  LUID  PrivilegeValue,
  KPROCESSOR_MODE  PreviousMode
  );
```

SeSinglePrivilegeCheck checks for the passed privilege value in the context of the current thread.

Parameters

PrivilegeValue

Specifies the value of the privilege being checked.

PreviousMode

Specifies the previous execution mode, one of **UserMode** or **KernelMode**.

Include

ntddk.h

Return Value

SeSinglePrivilegeCheck returns TRUE if the current subject has the required privilege.

Comments

Network transport drivers call this routine.

If *PreviousMode* is **KernelMode**, the privilege check always succeeds. Otherwise, this routine uses the token of the user-mode thread to determine whether the current (user-mode) thread has been granted the given privilege.

Callers of **SeSinglePrivilegeCheck** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlConvertLongToLuid, **RtlConvertUlongToLuid**, **RtlEqualLuid**, **SeValidSecurity-Descriptor**

SeValidSecurityDescriptor

```
BOOLEAN
  SeValidSecurityDescriptor(
  IN ULONG  Length,
  IN PSECURITY_DESCRIPTOR  SecurityDescriptor
  );
```

SeValidSecurityDescriptor returns whether a given security descriptor is structurally valid.

Parameters

Length

Specifies the size in bytes of the given security descriptor.

SecurityDescriptor

Points to the self-relative security descriptor, which must be buffered somewhere in system space.

Include

ntddk.h

Return Value

SeValidSecurityDescriptor returns TRUE if the buffered security descriptor is structurally valid.

Comments

SeValidSecurityDescriptor does not enforce policy. It simply checks that the given security descriptor data is formatted correctly. In particular, it checks the revision information, self relativity, owner, alignment, and, if available, SID, group, DACL, ACL, and/or SACL do not overflow the given *Length*. Consequently, callers of **SeValidSecurityDescriptor** cannot assume that a returned TRUE implies that the given security descriptor necessarily has valid contents.

If **SeValidSecurityDescriptor** returns TRUE, the given security descriptor can be passed on to another kernel-mode component because it is structurally valid. Otherwise, passing a structurally invalid security descriptor to be manipulated by another kernel-mode component can cause undefined results or even a system bugcheck.

To validate a security descriptor that was passed in from user mode, call **RtlValidSecurity-Descriptor** rather than **SeValidSecurityDescriptor**.

Callers of **SeValidSecurityDescriptor** must be running at IRQL PASSIVE_LEVEL.

See Also

RtlValidSecurityDescriptor

CHAPTER 11

ZwXxx Routines

Device and intermediate drivers might call some **Zw***Xxx* routines, which this chapter describes in compressed form: that is, information relevant to device and intermediate drivers is covered here.

References for the **Zw***Xxx* routines are in alphabetical order.

For an overview of the functionality of these routines, see Chapter 1, *Summary of Kernel-Mode Support Routines*.

ZwClose

```
NTSTATUS
  ZwClose(
  IN HANDLE  Handle
  );
```

ZwClose closes object handles. A named object is not actually deleted until all of its valid handles are closed and no referenced pointers remain.

Parameters

Handle

Is a valid handle for an open object.

Include

wdm.h or *ntddk.h*

Return Value

ZwClose can return one of the following:

STATUS_SUCCESS
STATUS_OBJECT_TYPE_MISMATCH
STATUS_ACCESS_DENIED
STATUS_INVALID_HANDLE

Comments

ZwClose is a generic routine that operates on any type of object.

Closing an open handle for an object causes that handle to become invalid. The reference count for the object handle is decremented and object retention checks are performed.

Any handle obtained from one of the **ZwCreate***Xxx* routines should eventually be released by calling **ZwClose**. Each such handle is created on a process-specific basis, so a driver must use each handle it creates in the context of a thread running in the appropriate process. For example, a handle returned by **ZwCreateKey** to a **DriverEntry** routine, which executes in a system process, cannot be subsequently used by the same driver's Dispatch*Xxx* routines, which usually execute either in the context of the thread issuing the current I/O request or, for lower-level drivers, in an arbitrary thread context.

Callers of **ZwClose** must be running at IRQL PASSIVE_LEVEL.

See Also

ZwCreateDirectoryObject, **ZwCreateFile**, **ZwCreateKey**, **ZwOpenKey**, **ZwOpenSection**

ZwCreateDirectoryObject

```
NTSTATUS
  ZwCreateDirectoryObject(
  OUT PHANDLE  DirectoryHandle,
  IN ACCESS_MASK  DesiredAccess,
  IN POBJECT_ATTRIBUTES  ObjectAttributes
  );
```

ZwCreateDirectoryObject creates or opens a directory object, which is a container for other objects.

Parameters

DirectoryHandle

Points to a variable that receives the directory object handle if the call is successful.

DesiredAccess

Specifies the type of access that the caller requires to the directory object. This value is compared with the granted access on an existing directory object. A caller can specify one or a combination of the following:

DesiredAccess Flags	Meaning
DIRECTORY_QUERY	Query access to the directory object
DIRECTORY_TRAVERSE	Name-lookup access to the directory object
DIRECTORY_CREATE_OBJECT	Name-creation access to the directory object
DIRECTORY_CREATE_SUBDIRECTORY	Subdirectory-creation access to the directory object
DIRECTORY_ALL_ACCESS	All of the preceding

ObjectAttributes

Points to a structure that specifies the object's attributes, which has already been initialized with **InitializeObjectAttributes**.

Include

wdm.h or *ntddk.h*

Return Value

ZwCreateDirectoryObject can return one of the following values:

STATUS_SUCCESS
STATUS_ACCESS_DENIED
STATUS_ACCESS_VIOLATION
STATUS_DATATYPE_MISALIGNMENT

Comments

A directory object is a container for other objects. Note that file system directories are *not* represented by directory objects, but rather by file objects.

Directory objects are an integral part of the system's object management and are manipulated indirectly as a result of other operations. For example, when a device object is created, its name is inserted in a directory object and the pointer counts of both the directory object and the named device object are incremented. Any named object's header contains a pointer to the directory object containing that object's name.

Drivers that create a set of device objects might set up a directory object when they initialize. For example, a disk driver might use this technique to group the device object representing a physical disk and the device objects representing partitions on that disk in a driver-created directory object.

Before the DriverEntry routine returns control, such a driver calls **ZwMakeTemporaryObject** if its directory object was initialized with the permanent attribute, and **ZwClose** to release the directory object created to hold such a group of related device objects.

If a directory object was initialized as temporary and its handle count becomes zero, the directory object's name is deleted. Name deletion occurs for a temporary object when the last handle to the object has been closed. A driver also can delete a directory object it creates when the object is no longer needed by using this technique.

Callers of **ZwCreateDirectoryObject** must be running at IRQL PASSIVE_LEVEL.

See Also

InitializeObjectAttributes, ObDereferenceObject, ZwClose, ZwMakeTemporaryObject

ZwCreateFile

```
NTSTATUS
  ZwCreateFile(
  OUT PHANDLE FileHandle,
  IN ACCESS_MASK DesiredAccess,
  IN POBJECT_ATTRIBUTES ObjectAttributes,
  OUT PIO_STATUS_BLOCK IoStatusBlock,
  IN PLARGE_INTEGER AllocationSize  OPTIONAL,
  IN ULONG FileAttributes,
  IN ULONG ShareAccess,
  IN ULONG CreateDisposition,
  IN ULONG CreateOptions,
  IN PVOID EaBuffer  OPTIONAL,
  IN ULONG EaLength
  );
```

ZwCreateFile either causes a new file or directory to be created, or it opens an existing file, device, directory, or volume, giving the caller a handle for the file object. This handle can be used by subsequent calls to manipulate data within the file or the file object's state or attributes. For example, a driver might call this routine during initialization to open a file of microcode for its device.

Parameters

FileHandle

Points to a variable that receives the file handle if the call is successful.

DesiredAccess

Specifies the type of access that the caller requires to the file or directory. The set of system-defined *DesiredAccess* flags determines the following specific access rights for file objects:

DesiredAccess Flags	Meaning
DELETE	The file can be deleted.
FILE_READ_DATA	Data can be read from the file.
FILE_READ_ATTRIBUTES	*FileAttributes* flags, described later, can be read.
FILE_READ_EA	Extended attributes associated with the file can be read. This flag is irrelevant to device and intermediate drivers.
READ_CONTROL	The access control list (ACL) and ownership information associated with the file can be read.
FILE_WRITE_DATA	Data can be written to the file.
FILE_WRITE_ATTRIBUTES	*FileAttributes* flags can be written.
FILE_WRITE_EA	Extended attributes (EAs) associated with the file can be written. This flag is irrelevant to device and intermediate drivers.
FILE_APPEND_DATA	Data can be appended to the file.
WRITE_DAC	The discretionary access control list (DACL) associated with the file can be written.
WRITE_OWNER	Ownership information associated with the file can be written.
SYNCHRONIZE	The returned *FileHandle* can be waited on to synchronize with the completion of an I/O operation.
FILE_EXECUTE	Data can be read into memory from the file using system paging I/O. This flag is irrelevant to device and intermediate drivers.

Callers of **ZwCreateFile** can specify one or a combination of the following, possibly ORed with additional compatible flags from the preceding **DesiredAccess Flags** list, for any file object that does not represent a directory file:

DesiredAccess to File Values	Maps to *DesiredAccess* Flags
GENERIC_READ	STANDARD_RIGHTS_READ, FILE_READ_DATA, FILE_READ_ATTRIBUTES, and FILE_READ_EA
GENERIC_WRITE	STANDARD_RIGHTS_WRITE, FILE_WRITE_DATA, FILE_WRITE_ATTRIBUTES, FILE_WRITE_EA, and FILE_APPEND_DATA
GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE, SYNCHRONIZE, and FILE_EXECUTE. This value is irrelevant to device and intermediate drivers.

The STANDARD_RIGHTS_*XXX* are predefined system values used to enforce security on system objects.

To open or create a directory file, as also indicated with the *CreateOptions* parameter, callers of **ZwCreateFile** can specify one or a combination of the following, possibly ORed with one or more compatible flags from the preceding **DesiredAccess Flags** list:

DesiredAccess to Directory Values	Meaning
FILE_LIST_DIRECTORY	Files in the directory can be listed.
FILE_TRAVERSE	The directory can be traversed: that is, it can be part of the pathname of a file.

The FILE_READ_DATA, FILE_WRITE_DATA, FILE_EXECUTE, and FILE_APPEND_DATA *DesiredAccess* flags are incompatible with creating or opening a directory file.

ObjectAttributes

Points to a structure already initialized with **InitializeObjectAttributes**. Members of this structure for a file object include the following:

Member	Value
ULONG Length	Specifies the number of bytes of *ObjectAttributes* data supplied. This value must be at least **sizeof** (OBJECT_ATTRIBUTES).
PUNICODE_STRING ObjectName	Points to a buffered Unicode string naming the file to be created or opened. This value must be a fully qualified file specification or the name of a device object, unless it is the name of a file relative to the directory specified by **RootDirectory**. For example, \Device\Floppy1\myfile.dat or \??\B:\myfile.dat could be the fully qualified file specification, provided that the floppy driver and overlying file system are already loaded. (Note: \?? replaces \DosDevices as the name of the Win32® object namespace. \DosDevices will still work, but \?? is translated faster by the object manager.)
HANDLE RootDirectory	Optionally specifies a handle to a directory obtained by a preceding call to **ZwCreateFile**. If this value is NULL, the **ObjectName** member must be a fully qualified file specification that includes the full path to the target file. If this value is nonNULL, the **ObjectName** member specifies a file name relative to this directory.

Member	Value
PSECURITY_DESCRIPTOR SecurityDescriptor	Optionally specifies a security descriptor to be applied to a file. ACLs specified by such a security descriptor are only applied to the file when it is created. If the value is NULL when a file is created, the ACL placed on the file is file-system-dependent; most file systems propagate some part of such an ACL from the parent directory file combined with the caller's default ACL. Device and intermediate drivers can set this member to NULL.
PSECURITY_QUALITY_OF_SERVICE SecurityQualityOfService	Specifies the access rights a server should be given to the client's security context. This value is nonNULL only when a connection to a protected server is established, allowing the caller to control which parts of the caller's security context are made available to the server and whether the server is allowed to impersonate the caller. Device and intermediate drivers usually set this member to NULL.
ULONG Attributes	Is a set of flags that controls the file object attributes. This value can be zero or OBJ_CASE_INSENSITIVE, which indicates that name-lookup code should ignore the case of **ObjectName** rather than performing an exact-match search. The value OBJ_INHERIT is irrelevant to device and intermediate drivers.

IoStatusBlock

Points to a variable that receives the final completion status and information about the requested operation. On return from **ZwCreateFile**, the **Information** member contains one of the following values:

FILE_CREATED
FILE_OPENED
FILE_OVERWRITTEN
FILE_SUPERSEDED
FILE_EXISTS
FILE_DOES_NOT_EXIST

AllocationSize

Optionally specifies the initial allocation size in bytes for the file. A nonzero value has no effect unless the file is being created, overwritten, or superseded.

FileAttributes

Explicitly specified attributes are applied only when the file is created, superseded, or, in some cases, overwritten. By default, this value is FILE_ATTRIBUTE_NORMAL, which

can be overridden by any other flag or by an ORed combination of compatible flags. Possible *FileAttributes* flags include the following:

FileAttributes Flags	Meaning
FILE_ATTRIBUTE_NORMAL	A file with standard attributes should be created.
FILE_ATTRIBUTE_READONLY	A read-only file should be created.
FILE_ATTRIBUTE_HIDDEN	A hidden file should be created.
FILE_ATTRIBUTE_SYSTEM	A system file should be created.
FILE_ATTRIBUTE_ARCHIVE	The file should be marked so that it will be archived.
FILE_ATTRIBUTE_TEMPORARY	A temporary file should be created.
FILE_ATTRIBUTE_ATOMIC_WRITE	An atomic-write file should be created. This flag is irrelevant to device and intermediate drivers.
FILE_ATTRIBUTE_XACTION_WRITE	A transaction-write file should be created. This flag is irrelevant to device and intermediate drivers.

ShareAccess

Specifies the type of share access that the caller would like to the file, as zero, or as one or a combination of the following:

ShareAccess Flags	Meaning
FILE_SHARE_READ	The file can be opened for read access by other threads' calls to **ZwCreateFile**.
FILE_SHARE_WRITE	The file can be opened for write access by other threads' calls to **ZwCreateFile**.
FILE_SHARE_DELETE	The file can be opened for delete access by other threads' calls to **ZwCreateFile**.

Device and intermediate drivers usually set *ShareAccess* to zero, which gives the caller exclusive access to the open file.

CreateDisposition

Specifies what to do, depending on whether the file already exists, as one of the following:

CreateDispostion Values	Meaning
FILE_SUPERSEDE	If the file already exists, replace it with the given file. If it does not, create the given file.
FILE_CREATE	If the file already exists, fail the request and do not create or open the given file. If it does not, create the given file.

CreateDispostion Values	Meaning
FILE_OPEN	If the file already exists, open it instead of creating a new file. If it does not, fail the request and do not create a new file.
FILE_OPEN_IF	If the file already exists, open it. If it does not, create the given file.
FILE_OVERWRITE	If the file already exists, open it and overwrite it. If it does not, fail the request.
FILE_OVERWRITE_IF	If the file already exists, open it and overwrite it. If it does not, create the given file.

CreateOptions

Specifies the options to be applied when creating or opening the file, as a compatible combination of the following flags:

CreateOptions Flags	Meaning
FILE_DIRECTORY_FILE	The file being created or opened is a directory file. With this flag, the *CreateDisposition* parameter must be set to one of FILE_CREATE, FILE_OPEN, or FILE_OPEN_IF. With this flag, other compatible *CreateOptions* flags include only the following: FILE_SYNCHRONOUS_IO_ALERT, FILE_SYNCHRONOUS_IO_NONALERT, FILE_WRITE_THROUGH, FILE_OPEN_FOR_BACKUP_INTENT, and FILE_OPEN_BY_FILE_ID.
FILE_NON_DIRECTORY_FILE	The file being opened must not be a directory file or this call will fail. The file object being opened can represent a data file, a logical, virtual, or physical device, or a volume.
FILE_WRITE_THROUGH	System services, FSDs, and drivers that write data to the file must actually transfer the data into the file before any requested write operation is considered complete. This flag is automatically set if the *CreateOptions* flag FILE_NO_INTERMEDIATE_BUFFERING is set.
FILE_SEQUENTIAL_ONLY	All accesses to the file will be sequential.
FILE_RANDOM_ACCESS	Accesses to the file can be random, so no sequential read-ahead operations should be performed on the file by FSDs or the system.

Continued

CreateOptions Flags	Meaning
FILE_NO_INTERMEDIATE_ BUFFERING	The file cannot be cached or buffered in a driver's internal buffers. This flag is incompatible with the *DesiredAccess* FILE_APPEND_DATA flag.
FILE_SYNCHRONOUS_IO_ALERT	All operations on the file are performed synchronously. Any wait on behalf of the caller is subject to premature termination from alerts. This flag also causes the I/O system to maintain the file position context. If this flag is set, the *DesiredAccess* SYNCHRONIZE flag also must be set.
FILE_SYNCHRONOUS_IO_NONALERT	All operations on the file are performed synchronously. Waits in the system to synchronize I/O queueing and completion are not subject to alerts. This flag also causes the I/O system to maintain the file position context. If this flag is set, the *Desired-Access* SYNCHRONIZE flag also must be set.
FILE_CREATE_TREE_CONNECTION	Create a tree connection for this file in order to open it over the network. This flag is irrelevant to device and intermediate drivers.
FILE_COMPLETE_IF_OPLOCKED	Complete this operation immediately with an alternate success code if the target file is oplocked, rather than blocking the caller's thread. If the file is oplocked, another caller already has access to the file over the network. This flag is irrelevant to device and intermediate drivers.
FILE_NO_EA_KNOWLEDGE	If the extended attributes on an existing file being opened indicate that the caller must understand EAs to properly interpret the file, fail this request because the caller does not understand how to deal with EAs. Device and intermediate drivers can ignore this flag.
FILE_DELETE_ON_CLOSE	Delete the file when the last handle to it is passed to **ZwClose**.
FILE_OPEN_BY_FILE_ID	The file name contains the name of a device and a 64-bit ID to be used to open the file. This flag is irrelevant to device and intermediate drivers.
FILE_OPEN_FOR_BACKUP_INTENT	The file is being opened for backup intent, hence, the system should check for certain access rights and grant the caller the appropriate accesses to the file before checking the input *DesiredAccess* against the file's security descriptor. This flag is irrelevant to device and intermediate drivers.

EaBuffer

For device and intermediate drivers, this parameter must be a NULL pointer.

EaLength

For device and intermediate drivers, this parameter must be zero.

Include

wdm.h or *ntddk.h*

Return Value

ZwCreateFile either returns STATUS_SUCCESS or an appropriate error status. If it returns an error status, the caller can find more information about the cause of the failure by checking the *IoStatusBlock*.

Comments

There are two alternate ways to specify the name of the file to be created or opened with **ZwCreateFile**:

1. As a fully qualified pathname, supplied in the **ObjectName** member of the input *ObjectAttributes*
2. As pathname relative to the directory file represented by the handle in the **RootDirectory** member of the input *ObjectAttributes*

Certain *DesiredAccess* flags and combinations of flags have the following effects:

- For a caller to synchronize an I/O completion by waiting on the returned *FileHandle*, the SYNCHRONIZE flag must be set. Otherwise, a caller that is a device or intermediate driver must synchronize an I/O completion by using an event object.
- If only the FILE_APPEND_DATA and SYNCHRONIZE flags are set, the caller can write only to the end of the file, and any offset information on writes to the file is ignored. However, the file will automatically be extended as necessary for this type of write operation.
- Setting the FILE_WRITE_DATA flag for a file also allows writes beyond the end of the file to occur. The file is automatically extended for this type of write, as well.
- If only the FILE_EXECUTE and SYNCHRONIZE flags are set, the caller cannot directly read or write any data in the file using the returned *FileHandle*: that is, all operations on the file occur through the system pager in response to instruction and data accesses. Device and intermediate drivers should not set the FILE_EXECUTE flag in *DesiredAccess*.

The *ShareAccess* parameter determines whether separate threads can access the same file, possibly simultaneously. Provided that both file openers have the privilege to access a file in the specified manner, the file can be successfully opened and shared. If the original caller of **ZwCreateFile** does not specify FILE_SHARE_READ, FILE_SHARE_WRITE, or FILE_SHARE_DELETE, no other open operations can be performed on the file: that is, the original caller is given exclusive access to the file.

In order for a shared file to be successfully opened, the requested *DesiredAccess* to the file must be compatible with both the *DesiredAccess* and *ShareAccess* specifications of all preceding opens that have not yet been released with **ZwClose**. That is, the *DesiredAccess* specified to **ZwCreateFile** for a given file must not conflict with the accesses that other openers of the file have disallowed.

The *CreateDispostion* value FILE_SUPERSEDE requires that the caller have DELETE access to a existing file object. If so, a successful call to **ZwCreateFile** with FILE_SUPERSEDE on an existing file effectively deletes that file, and then recreates it. This implies that, if the file has already been opened by another thread, it opened the file by specifying a *ShareAccess* parameter with the FILE_SHARE_DELETE flag set. Note that this type of disposition is consistent with the POSIX style of overwriting files.

The *CreateDisposition* values FILE_OVERWRITE_IF and FILE_SUPERSEDE are similar. If **ZwCreateFile** is called with a existing file and either of these *CreateDisposition* values, the file will be replaced.

Overwriting a file is semantically equivalent to a supersede operation, except for the following:

- The caller must have write access to the file, rather than delete access. This implies that, if the file has already been opened by another thread, it opened the file with the FILE_SHARE_WRITE flag set in the input *ShareAccess*.
- The specified file attributes are logically ORed with those already on the file. This implies that, if the file has already been opened by another thread, a subsequent caller of **ZwCreateFile** cannot disable existing *FileAttributes* flags but can enable additional flags for the same file. Note that this style of overwriting files is consistent with MS-DOS®, Windows® 3.1, and with OS/2.

The *CreateOptions* FILE_DIRECTORY_FILE value specifies that the file to be created or opened is a directory file. When a directory file is created, the file system creates an appropriate structure on the disk to represent an empty directory for that particular file system's on-disk structure. If this option was specified and the given file to be opened is not a directory file, or if the caller specified an inconsistent *CreateOptions* or *CreateDispostion* value, the call to **ZwCreateFile** will fail.

The *CreateOptions* FILE_NO_INTERMEDIATE_BUFFERING flag prevents the file system from performing any intermediate buffering on behalf of the caller. Specifying this value places certain restrictions on the caller's parameters to other **Zw..File** routines, including the following:

- Any optional *ByteOffset* passed to **ZwReadFile** or **ZwWriteFile** must be an integral of the sector size.
- The *Length* passed to **ZwReadFile** or **ZwWriteFile**, must be an integral of the sector size. Note that specifying a read operation to a buffer whose length is exactly the sector size might result in a lesser number of significant bytes being transferred to that buffer if the end of the file was reached during the transfer.
- Buffers must be aligned in accordance with the alignment requirement of the underlying device. This information can be obtained by calling **ZwCreateFile** to get a handle for the file object that represents the physical device, and, then, calling **ZwQueryInformationFile** with that handle. For a list of the system FILE_XXX_ALIGNMENT values, see *DEVICE_OBJECT* in Chapter 12.
- Calls to **ZwSetInformationFile** with the *FileInformationClass* parameter set to **FilePositionInformation** must specify an offset that is an integral of the sector size.

The *CreateOptions* FILE_SYNCHRONOUS_IO_ALERT and FILE_SYNCHRONOUS_IO_NONALERT, which are mutually exclusive as their names suggest, specify that all I/O operations on the file are to be synchronous as long as they occur through the file object referred to by the returned *FileHandle*. All I/O on such a file is serialized across all threads using the returned handle. With either of these *CreateOptions*, the *DesiredAccess* SYNCHRONIZE flag must be set so that the I/O Manager will use the file object as a synchronization object. With either of these *CreateOptions* set, the I/O Manager maintains the "file position context" for the file object, an internal, current file position offset. This offset can be used in calls to **ZwReadFile** and **ZwWriteFile**. Its position also can be queried or set with **ZwQueryInformationFile** and **ZwSetInformationFile**.

Callers of **ZwCreateFile** must be running at IRQL PASSIVE_LEVEL.

See Also

InitializeObjectAttributes, DEVICE_OBJECT, IO_STATUS_BLOCK, **ZwClose**, **ZwReadFile**, **ZwQueryInformationFile**, **ZwSetInformationFile**, **ZwWriteFile**

ZwCreateKey

```
NTSTATUS
  ZwCreateKey(
  OUT PHANDLE  KeyHandle,
  IN ACCESS_MASK  DesiredAccess,
  IN POBJECT_ATTRIBUTES  ObjectAttributes,
  IN ULONG  TitleIndex,
  IN PUNICODE_STRING  Class  OPTIONAL,
  IN ULONG  CreateOptions,
  OUT PULONG  Disposition  OPTIONAL
  );
```

ZwCreateKey opens an existing key or creates a new key in the registry.

Parameters

KeyHandle

Points to a returned handle for a newly created or existing key if this call is successful.

DesiredAccess

Specifies the type of access that the caller requires to the key. The set of system-defined *DesiredAccess* flags determines the following specific access rights for key objects:

DesiredAccess Flags	Meaning
KEY_QUERY_VALUES	Value entries for the key can be read.
KEY_SET_VALUE	Value entries for the key can be written.
KEY_CREATE_SUB_KEY	Subkeys for the key can be created.
KEY_ENUMERATE_SUB_KEYS	All subkeys for the key can be read.
KEY_NOTIFY	This flag is irrelevant to device and intermediate drivers, and to other kernel-mode code.
KEY_CREATE_LINK	A symbolic link to the key can be created. This flag is irrelevant to device and intermediate drivers.

Callers of **ZwCreateKey** can specify one or a compatible combination of the following for any key object:

DesiredAccess to Key Values	Maps to *DesiredAccess* Flags
KEY_READ	STANDARD_RIGHTS_READ, KEY_QUERY_VALUES, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY
KEY_WRITE	STANDARD_RIGHTS_WRITE, KEY_SET_VALUE, and KEY_CREATE_SUBKEY

DesiredAccess to Key Values	Maps to *DesiredAccess* Flags
KEY_EXECUTE	KEY_READ. This value is irrelevant to device and intermediate drivers.
KEY_ALL_ACCESS	STANDARD_RIGHTS_ALL, KEY_QUERY_VALUES, KEY_SET_VALUE, KEY_CREATE_SUB_KEY, KEY_ENUMERATE_SUBKEY, KEY_NOTIFY and KEY_CREATE_LINK

The STANDARD_RIGHTS_*XXX* are predefined system values used to enforce security on system objects.

ObjectAttributes

Points to the initialized object attributes of the key being opened or created. An **ObjectName** string for the key must be specified. If a **RootDirectory** handle also is supplied, the given name is relative to the key represented by the handle. Any given name must be within the object name space allocated to the registry, meaning that all names must begin with **\Registry**. **RootHandle,** if present, must be a handle to the root directory object, to **\Registry**, or to a key under **\Registry**.

TitleIndex

Device and intermediate drivers should set this parameter to zero.

Class

Points to the object class of the key. To the Configuration Manager, this is just a Unicode string.

CreateOptions

Specifies options to be applied when creating a key, as a compatible combination of the following:

Value	Meaning
REG_OPTION_VOLATILE	Key is not to be stored across boots.
REG_OPTION_NON_VOLATILE	Key is preserved when the system is rebooted.
REG_OPTION_CREATE_LINK	The created key is a symbolic link. This value is irrelevant to device and intermediate drivers.
REG_OPTION_BACKUP_RESTORE	Key is being opened or created with special privileges allowing backup/restore operations. This value is irrelevant to device and intermediate drivers.

Disposition

Points to a variable that receives a value indicating whether a new key was created in the **\Registry** tree or an existing one opened:

Value	Meaning
REG_CREATED_NEW_KEY	A new key object was created.
REG_OPENED_EXISTING_KEY	An existing key object was opened.

Include

wdm.h or *ntddk.h*

Return Value

ZwCreateKey returns STATUS_SUCCESS if the given key was created or opened.

Comments

If the key specified by *ObjectAttributes* does not exist, an attempt is made to create it. For this attempt to succeed, the new key must be a direct subkey of the key referred to by *KeyHandle*, and the given *KeyHandle* must have been opened for KEY_CREATE_SUB_KEY access.

If the specified key already exists, it is opened and its value is not affected in any way.

The security attributes specified by *ObjectAttributes* when a key is created determine whether the specified *DesiredAccess* is granted on subsequent calls to **ZwCreateKey** and **ZwOpenKey**.

Callers of **ZwCreateKey** must be running at IRQL PASSIVE_LEVEL.

See Also

InitializeObjectAttributes, **ZwClose**, **ZwDeleteKey**, **ZwEnumerateKey**, **ZwEnumerateValueKey**, **ZwFlushKey**, **ZwOpenKey**, **ZwQueryValueKey**, **ZwSetValueKey**

ZwDeleteKey

```
NTSTATUS
  ZwDeleteKey(
  IN HANDLE  KeyHandle
  );
```

ZwDeleteKey deletes an open key from the registry.

Parameters

KeyHandle

Is a handle returned by a successful call to **ZwCreateKey** or **ZwOpenKey**.

Include

wdm.h or *ntddk.h*

Return Value

ZwDeleteKey can return one of the following values:

STATUS_SUCCESS
STATUS_ACCESS_DENIED
STATUS_INVALID_HANDLE

Comments

The key must have been opened for DELETE access for a deletion to succeed; the *Desired-Access* value KEY_ALL_ACCESS includes DELETE access. The actual storage for the key is deleted when the last handle to the key is closed.

A call to **ZwDeleteKey** causes the *KeyHandle* to become invalid.

Callers of **ZwDeleteKey** must be running at IRQL PASSIVE_LEVEL.

See Also

ZwClose, **ZwCreateKey**, **ZwOpenKey**

ZwEnumerateKey

```
NTSTATUS
  ZwEnumerateKey(
  IN HANDLE  KeyHandle,
  IN ULONG  Index,
  IN KEY_INFORMATION_CLASS  KeyInformationClass,
  OUT PVOID  KeyInformation,
  IN ULONG  Length,
  OUT PULONG  ResultLength
  );
```

ZwEnumerateKey returns information about the subkeys of an open key.

Parameters

KeyHandle

Is the handle, returned by a successful call to **ZwCreateKey** or **ZwOpenKey**, of the key whose subkeys are to be enumerated.

Index

Specifies the zero-based index of the subkey for which the information is requested.

KeyInformationClass

Specifies the type of information returned in the *KeyInformation* buffer as one of the following system-defined values:

KeyBasicInformation
KeyNodeInformation
KeyFullInformation

KeyInformation

Points to a caller-allocated buffer to receive the requested data.

Length

Is the size in bytes of the *KeyInformation* buffer, which the caller should set according to the given *KeyInformationClass*.

ResultLength

Points to the number of bytes actually returned to *KeyInformation* or, if the input *Length* is too small, points to the number of bytes required for the available information.

Include

wdm.h or *ntddk.h*

Return Value

ZwEnumerateKey returns STATUS_SUCCESS, together with the name of the *Index* subkey to the given *KeyInformation* buffer. Otherwise, **ZwEnumerateKey** can return one of the following:

STATUS_NO_MORE_ENTRIES
STATUS_BUFFER_TOO_SMALL

Comments

The *KeyHandle* passed to **ZwEnumerateKey** must have been opened with the KEY_ENUMERATE_SUB_KEY *DesiredAccess* flag set for this call to succeed. See **ZwCreateKey** for a description of possible values for *DesiredAccess*.

The *Index* parameter is simply a way to select among subkeys of the key referred to by the *KeyHandle*. Two calls to **ZwEnumerateKey** with the same *Index* are not guaranteed to return the same result.

Note that callers of the **Rtl..Registry** routines are required to provide the name of the key. Drivers can call **ZwEnumerateKey** to get unknown names of the subkeys for a key with a known name.

Callers of **ZwEnumerateKey** must be running at IRQL PASSIVE_LEVEL.

See Also

KEY_BASIC_INFORMATION, KEY_FULL_INFORMATION, KEY_NODE_INFORMATION, **RtlCheckRegistryKey**, **RtlCreateRegistryKey**, **RtlDeleteRegistryValue**, **RtlQueryRegistryValues**, **RtlWriteRegistryValue**, **ZwCreateKey**, **ZwEnumerateValueKey**, **ZwOpenKey**

ZwEnumerateValueKey

```
NTSTATUS
  ZwEnumerateValueKey(
  IN HANDLE  KeyHandle,
  IN ULONG  Index,
  IN KEY_VALUE_INFORMATION_CLASS  KeyValueInformationClass,
  OUT PVOID  KeyValueInformation,
  IN ULONG  Length,
  OUT PULONG  ResultLength
  );
```

ZwEnumerateValueKey returns information about the value entries of an open key.

Parameters

KeyHandle

Is the handle, returned by a successful call to **ZwCreateKey** or **ZwOpenKey**, of the key whose value entries are to be enumerated.

Index

Specifies the zero-based index of a subkey for which the value information is requested.

KeyValueInformationClass

Specifies the type of information returned in the *KeyValueInformation* buffer as one of the following:

KeyValueBasicInformation
KeyValueFullInformation
KeyValuePartialInformation

KeyValueInformation

Points to a caller-allocated buffer to receive the requested data.

Length

Is the size in bytes of the *KeyValueInformation* buffer, which the caller should set according to the given *KeyValueInformationClass*.

ResultLength

Points to number of bytes actually returned to *KeyValueInformation* or, if the input *Length* is too small, points to the number of bytes required for the available information.

Include

wdm.h or *ntddk.h*

Return Value

ZwEnumerateValueKey returns STATUS_SUCCESS, together with the name of the *Index* subkey to the given *KeyValueInformation* buffer. Otherwise, **ZwEnumerateValueKey** can return one of the following:

STATUS_NO_MORE_VALUES
STATUS_BUFFER_TOO_SMALL
STATUS_INVALID_PARAMETER

Comments

The *KeyHandle* passed to **ZwEnumerateValueKey** must have been opened with the KEY_QUERY_VALUES *DesiredAccess* flag set for this call to succeed. See **ZwCreateKey** for a description of possible values for *DesiredAccess*.

The *Index* is simply a way to select among subkeys with value entries. Two calls to **ZwEnumerateValueKey** with the same *Index* are not guaranteed to return the same results.

Callers of **ZwEnumerateValueKey** must be running at IRQL PASSIVE_LEVEL.

See Also

ZwClose, **ZwCreateKey**, **ZwOpenKey**, **ZwQueryValueKey**

ZwFlushKey

```
NTSTATUS
  ZwFlushKey(
  IN HANDLE  KeyHandle
  );
```

ZwFlushKey forces a registry key to be committed to disk.

Parameters

KeyHandle

Is the handle, returned by a successful call to **ZwCreateKey** or **ZwOpenKey**, of the key to be flushed.

Include

ntddk.h

Return Value

ZwFlushKey returns STATUS_SUCCESS if the key information was transferred to disk.

Comments

Changes made by **ZwCreateKey** or **ZwSetValueKey** can be flushed to disk with **ZwFlushKey**. This routine does not return to its caller until any changed data associated with the given *KeyHandle* has been written to permanent store.

Note This routine can flush the entire registry. Accordingly, it can generate a great deal of I/O. Since the system automatically flushes key changes every few seconds, it is seldom necessary to call ZwFlushKey.

Callers of **ZwFlushKey** must be running at IRQL PASSIVE_LEVEL.

See Also

ZwCreateKey, **ZwOpenKey**, **ZwSetValueKey**

ZwMakeTemporaryObject

```
NTSTATUS
  ZwMakeTemporaryObject(
  IN HANDLE  Handle
  );
```

ZwMakeTemporaryObject changes the attributes of an object to make it temporary.

Parameters

Handle

Specifies an open handle for an object.

Include

wdm.h or *ntddk.h*

Return Value

ZwMakeTemporaryObject can return one of the following:

STATUS_SUCCESS
STATUS_ACCESS_DENIED
STATUS_INVALID_HANDLE

Comments

ZwMakeTemporaryObject is a generic routine that operates on any type of object.

Making an object temporary causes the permanent flag of the associated object to be cleared. A temporary object has a name only as long as its handle count is greater than zero. When the handle count reaches zero, the system deletes the object name and adjusts the pointer count for the object appropriately.

Callers of **ZwMakeTemporaryObject** must be running at IRQL PASSIVE_LEVEL.

See Also

InitializeObjectAttributes, ZwClose, ZwCreateDirectoryObject, ZwCreateFile

ZwMapViewOfSection

```
NTSTATUS
  ZwMapViewOfSection(
  IN HANDLE  SectionHandle,
  IN HANDLE  ProcessHandle,
  IN OUT PVOID  *BaseAddress,
  IN ULONG  ZeroBits,
  IN ULONG  CommitSize,
  IN OUT PLARGE_INTEGER  SectionOffset  OPTIONAL,
  IN OUT PSIZE_T  ViewSize,
  IN SECTION_INHERIT  InheritDisposition,
  IN ULONG  AllocationType,
  IN ULONG  Protect
  );
```

ZwMapViewOfSection maps a view of a section into the virtual address space of a subject process.

Parameters

SectionHandle

Is the handle returned by a successful call to **ZwOpenSection**.

ProcessHandle

Is the handle of an opened process object, representing the process for which the view should be mapped.

BaseAddress

Points to a variable that will receive the base address of the view. If the initial value of this argument is nonNULL, the view is allocated starting at the specified virtual address rounded down to the next 64-kilobyte address boundary.

ZeroBits

Specifies the number of high-order address bits that must be zero in the base address of the section view. The value of this argument must be less than 21 and is used only when the operating system determines where to allocate the view, as when *BaseAddress* is NULL.

CommitSize

Specifies the size, in bytes, of the initially committed region of the view. *CommitSize* is only meaningful for page-file backed sections. For mapped sections, both data and image are always committed at section creation time. This parameter is ignored for mapped files. This value is rounded up to the next host-page-size boundary.

SectionOffset

Points to the offset, in bytes, from the beginning of the section to the view. If this pointer is nonNULL, the given value is rounded down to the next allocation granularity size boundary.

ViewSize

Points to a variable that will receive the actual size, in bytes, of the view. If the value of this parameter is zero, a view of the section will be mapped starting at the specified section offset and continuing to the end of the section. Otherwise, the initial value of this argument specifies the size of the view, in bytes, and is rounded up to the next host page-size boundary.

InheritDispostion

Specifies how the view is to be shared by a child process created with a create process operation. Device and intermediate drivers should set this parameter to zero.

AllocationType

A set of flags that describes the type of allocation to be performed for the specified region of pages.

Protect

Specifies the protection for the region of initially committed pages. Device and intermediate drivers should set this value to PAGE_READWRITE.

Include

wdm.h or *ntddk.h*

Return Value

ZwMapViewOfSection can return one of the following:

STATUS_SUCCESS
STATUS_ACCESS_DENIED
STATUS_INVALID_HANDLE

Comments

Several different views of a section can be concurrently mapped into the virtual address space of a process. Likewise, several different views of a section can be concurrently mapped into the virtual address space of several processes.

If the specified section does not exist or the access requested is not allowed, **ZwMapViewOfSection** returns an error.

Callers of **ZwMapViewOfSection** must be running at IRQL PASSIVE_LEVEL.

See Also

ZwOpenSection, **ZwUnmapViewOfSection**

ZwOpenFile

```
NTSTATUS
ZwOpenFile(
    OUT PHANDLE FileHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN ULONG ShareAccess,
    IN ULONG OpenOptions
    );
```

ZwOpenFile opens an existing file, device, directory, or volume, and returns a handle for the file object.

Parameters

FileHandle

Pointer to a handle for the opened file.

DesiredAccess

Specifies the type of required access to the file.

ObjectAttributes

Pointer to a structure that a caller initializes with **InitializeObjectAttributes**.

IoStatusBlock

Pointer to a structure that contains information about the requested operation and the final completion status.

ShareAccess

Specifies the type of share access for the file.

OpenOptions

Specifies the options to be applied when opening the file.

Include

ntddk.h

Return value

ZwOpenFile either returns STATUS_SUCCESS or an appropriate error status. If it returns an error status, the caller can get more information about the error by checking status information returned in *IoStatusBlock*.

Comments

ZwOpenFile provides a subset of the functionality provided by **ZwCreateFile**.

See Also

InitializeObjectAttributes, **ZwCreateFile**

ZwOpenKey

```
NTSTATUS
  ZwOpenKey(
  OUT PHANDLE  KeyHandle,
  IN ACCESS_MASK  DesiredAccess,
  IN POBJECT_ATTRIBUTES  ObjectAttributes
  );
```

ZwOpenKey opens an existing key in the registry.

Parameters

KeyHandle

Points to a returned handle for the key specified in *ObjectAttributes* if this call is successful.

DesiredAccess

Specifies the access rights desired to the key. See **ZwCreateKey** for a description of possible values for this parameter.

ObjectAttributes

Points to the initialized object attributes of the key being opened. See the description of **ZwCreateKey** for more information.

Include

wdm.h or *ntddk.h*

Return Value

ZwOpenKey returns STATUS_SUCCESS if the given key was opened. Otherwise, it can return an error status, including the following:

STATUS_INVALID_HANDLE
STATUS_ACCESS_DENIED

Comments

ZwOpenKey or **ZwCreateKey** must be called before any of the **Zw...Key** routines that require an input *KeyHandle*.

If the specified key does not exist or the *DesiredAccess* requested is not allowed, **ZwOpenKey** returns an error status, and the *KeyHandle* remains invalid.

ZwOpenKey ignores the security information in the input *ObjectAttributes*. Access rights for a key object can be set only when the key is created.

Callers of **ZwOpenKey** must be running at IRQL PASSIVE_LEVEL.

See Also

InitializeObjectAttributes, ZwCreateKey, ZwDeleteKey, ZwEnumerateKey, ZwEnumerateValueKey, ZwFlushKey, ZwQueryKey, ZwQueryValueKey, ZwSetValueKey

ZwOpenSection

```
NTSTATUS
  ZwOpenSection(
  OUT PHANDLE  SectionHandle,
  IN ACCESS_MASK  DesiredAccess,
  IN POBJECT_ATTRIBUTES  ObjectAttributes
  );
```

ZwOpenSection opens a handle for an existing section object.

Parameters

SectionHandle

Points to a variable that will receive the section object handle if this call is successful.

DesiredAccess

Specifies a mask representing the requested access to the object. The set of system-defined *DesiredAccess* flags relevant to device and intermediate drivers are the following:

DesiredAccess Flags	Meaning
SECTION_MAP_WRITE	A mapped view can be written.
SECTION_MAP_READ	A mapped view can be read.

A caller can specify SECTION_ALL_ACCESS, which sets all of the defined flags ORed with the system-defined STANDARD_RIGHTS_REQUIRED.

ObjectAttributes

Points to the initialized object attributes of the section to be opened.

Include

wdm.h or *ntddk.h*

Return Value

ZwOpenSection can return one of the following:

STATUS_SUCCESS
STATUS_ACCESS_DENIED
STATUS_INVALID_HANDLE

Comments

If the specified section does not exist or the access requested is not allowed, the operation fails.

Callers of **ZwOpenSection** must be running at PASSIVE_LEVEL.

See Also

InitializeObjectAttributes, **ZwMapViewOfSection**, **ZwUnmapViewOfSection**

ZwOpenSymbolicLinkObject

```
NTSTATUS
    ZwOpenSymbolicLinkObject(
        OUT PHANDLE LinkHandle,
        IN ACCESS_MASK DesiredAccess,
        IN POBJECT_ATTRIBUTES ObjectAttributes
        );
```

ZwOpenSymbolicLinkObject returns a handle to an existing symbolic link.

Parameters

LinkHandle

Points to a returned handle for the symbolic link object specified in *ObjectAttributes* if the call was successful.

DesiredAccess

Specifies the type of access that the caller requires to the key. This is most commonly GENERIC_READ access such that the returned handle can be used with **ZwQuerySymbolicLinkObject**.

ObjectAttributes

Points to the initialized object attributes for the symbolic link being opened. An **ObjectName** string for the symbolic link must be specified.

Include

ntddk.h

Return Value

ZwOpenSymbolicLinkObject returns STATUS_SUCCESS if the symbolic link was opened.

Comments

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

InitializeObjectAttributes, **ZwQuerySymbolicLinkObject**

ZwQueryInformationFile

```
NTSTATUS
  ZwQueryInformationFile(
  IN HANDLE  FileHandle,
  OUT PIO_STATUS_BLOCK  IoStatusBlock,
  OUT PVOID  FileInformation,
  IN ULONG  Length,
  IN FILE_INFORMATION_CLASS  FileInformationClass
  );
```

ZwQueryInformationFile returns various kinds of information about a given file object.

Parameters

FileHandle

Is the handle returned by a successful call to **ZwCreateFile**.

IoStatusBlock

Points to a variable that receives the final completion status and information about the operation.

FileInformation

Points to a caller-allocated buffer or variable that receives the desired information about the file. The contents of *FileInformation* are defined by the *FileInformationClass* parameter, described later.

Length

Specifies the size in bytes of *FileInformation*, which the caller should set according to the given *FileInformationClass*.

FileInformationClass

Specifies the type of information to be returned about the file. Device and intermediate drivers can specify any of the following:

FileInformationClass Value	Meaning
FileBasicInformation	Return FILE_BASIC_INFORMATION about the file. The caller must have opened the file with the *DesiredAccess* FILE_READ_ATTRIBUTES flag set.
FileStandardInformation	Return FILE_STANDARD_INFORMATION about the file. The caller can query this information as long as the file is open, without any particular requirements for *DesiredAccess*.
FilePositionInformation	Return FILE_POSITION_INFORMATION about the file. The caller must have opened the file with the *DesiredAccess* FILE_READ_DATA or FILE_WRITE_DATA flag set and with either of the *CreateOptions* FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT.
FileAlignmentInformation	Return FILE_ALIGNMENT_INFORMATION about the file. The caller can query this information as long as the file is open, without any particular requirements for *DesiredAccess*. This information is useful if the file was opened with the *CreateOptions* FILE_NO_INTERMEDIATE_BUFFERING flag set.
FileNameInformation	Return FILE_NAME_INFORMATION about the file. This might include the full file path or only a portion of the path. See the comments below for details on the file name syntax.

Include

wdm.h or *ntddk.h*

Return Value

ZwQueryInformationFile returns STATUS_SUCCESS or an appropriate error status. It also returns the number of bytes actually written to the given *FileInformation* buffer in the **Information** member of *IoStatusBlock*.

Comments

ZwQueryInformationFile returns information about the given file. Note that it returns zero in any member of a FILE_*XXX*_INFORMATION structure that is not supported by a particular device or file system. For example, the FAT file system does not support file-creation times, so **ZwQueryInformationFile** sets the **CreationTime** member of returned FILE_BASIC_INFORMATION to zero for files on a FAT partition.

Callers of **ZwQueryInformationFile** must be running at IRQL PASSIVE_LEVEL.

When *FileInformationClass* equals **FileNameInformation**, the file name is returned in the FILE_NAME_INFORMATION structure. The precise syntax of the file name depends on a number of factors:

If the file was opened by submitting a full path and file name to **ZwCreateFile**, then **ZwQueryInformationFile** returns that full path and file name.

If the **ObjectAttributes->RootDirectory** handle was opened by name in a call to **ZwCreateFile**, and subsequently the file was opened by **ZwCreateFile** relative to this root directory handle, then the full path and file name are returned.

If the **ObjectAttributes->RootDirectory** handle was opened by file ID (using the FILE_OPEN_BY_FILE_ID flag) in a call to **ZwCreateFile**, and subsequently the file was opened by **ZwCreateFile** relative to this root directory handle, then only the relative path will be returned.

However, if the user has BYPASS_TRAVERSE_PRIVILEGE, the full path and file name will be returned in all cases.

If only the relative path is returned, the file name string will not begin with a backslash.

If the full path and file name are returned, the string will begin with a single backslash, regardless of its location. Thus the file *C:\dir1\dir2\filename.ext* will appear as *\dir1\dir2\filename.ext*, while the file *\\server\share\dir1\dir2\filename.ext* will appear as *\server\share\dir1\dir2\filename.ext*.

See Also

FILE_ALIGNMENT_INFORMATION, FILE_BASIC_INFORMATION, FILE_NAME_INFORMATION, FILE_POSITION_INFORMATION, FILE_STANDARD_INFORMATION, **ZwCreateFile**, **ZwSetInformationFile**

ZwQueryKey

```
NTSTATUS
  ZwQueryKey(
  IN HANDLE  KeyHandle,
  IN KEY_INFORMATION_CLASS  KeyInformationClass,
    OUT PVOID  KeyInformation,
  IN ULONG  Length,
  OUT PULONG  ResultLength
  );
```

ZwQueryKey provides data about the class of a key, and the number and sizes of its subkeys.

Parameters

KeyHandle

Is the handle, returned by a successful call to **ZwCreateKey** or **ZwOpenKey**, of the key to be queried.

KeyInformationClass

Specifies the type of information returned in the buffer as one of the following:

KeyBasicInformation
KeyFullInformation
KeyNodeInformation

KeyInformation

Points to a caller-allocated buffer to receive the requested data.

Length

Is the size in bytes of the *KeyInformation* buffer, which the caller should set according to the given *KeyInformationClass*.

ResultLength

Points to number of bytes actually returned to *KeyInformation* or, if the input *Length* is too small, points to the number of bytes required for the available information.

Include

wdm.h or *ntddk.h*

Return Value

ZwQueryKey returns STATUS_SUCCESS if it returned the requested information in the *KeyInformation* buffer. Otherwise, **ZwQueryKey** can return one of the following:

STATUS_BUFFER_TOO_SMALL
STATUS_INVALID_PARAMETER

Comments

The *KeyHandle* passed to **ZwQueryKey** must have been opened with the KEY_QUERY_KEY *DesiredAccess* flag set for this call to succeed. See **ZwCreateKey** for a description of possible values for *DesiredAccess*.

ZwQueryKey returns information about the size of the value entries, the number of subkeys, the length of their names, and the size of their value entries that its caller can use to allocate buffers for registry data.

For example, a successful caller of **ZwQueryKey** might allocate a buffer for a subkey, call **ZwEnumerateKey** to get the name of the subkey, and pass that name to an **Rtl..Registry** routine.

Callers of **ZwQueryKey** must be running at IRQL PASSIVE_LEVEL.

See Also

KEY_BASIC_INFORMATION, KEY_FULL_INFORMATION, KEY_NODE_INFORMATION, **ZwClose**, **ZwEnumerateKey**, **ZwOpenKey**

ZwQuerySymbolicLinkObject

```
NTSTATUS
    ZwQuerySymbolicLinkObject(
        IN HANDLE LinkHandle,
        IN OUT PUNICODE_STRING LinkTarget,
        OUT PULONG ReturnedLength OPTIONAL
        );
```

ZwQuerySymbolicLinkObject returns a Unicode string containing the target of the symbolic link.

Parameters

LinkHandle

Specifies a valid handle to an open symbolic link object obtained by calling **ZwOpenSymbolicLinkObject**.

LinkTarget

Points to an initialized Unicode string that contains the target of the symbolic link, specified by *LinkHandle*, if the call was successful.

ReturnedLength

Optionally, points to a unsigned long integer that on input contains the maximum number of bytes to copy into the Unicode string at *LinkTarget*. On output, the unsigned long integer contains the length of the Unicode string naming the target of the symbolic link.

Include

ntddk.h

Return Value

ZwOpenSymbolicLinkObject returns either STATUS_SUCCESS to indicate the routine completed without error or STATUS_BUFFER_TOO_SMALL if the Unicode string provided at *LinkTarget* is too small to hold the returned string.

Comments

Before calling this routine, driver writers must ensure that the Unicode string at *LinkTarget* has been properly initialized and a buffer for the string has been allocated. The **MaximumLength** and **Buffer** members of the Unicode string must be set before calling **ZwQuerySymbolicLinkObject** or the call will fail.

If **ZwQuerySymbolicLinkObject** returns STATUS_BUFFER_TOO_SMALL drivers should examine the value returned at *ReturnedLength*. The number returned in this variable indicates the maximum length that the Unicode string for the target of the symbolic link.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

ZwOpenSymbolicLinkObject

ZwQueryValueKey

```
NTSTATUS
  ZwQueryValueKey(
  IN HANDLE  KeyHandle,
  IN PUNICODE_STRING  ValueName,
  IN KEY_VALUE_INFORMATION_CLASS  KeyValueInformationClass,
  OUT PVOID  KeyValueInformation,
  IN ULONG  Length,
  OUT PULONG  ResultLength
  );
```

ZwQueryValueKey returns the value entries for an open registry key.

Parameters

KeyHandle

Is the handle, returned by a successful call to **ZwCreateKey** or **ZwOpenKey**, of key for which value entries are to be read.

ValueName

Points to the name of the value entry for which the data is requested.

KeyValueInformationClass

Specifies the type of information requested as one of the following:

KeyValueBasicInformation
KeyValueFullInformation
KeyValuePartialInformation

KeyValueInformation

Points to a caller-allocated buffer to receive the requested data.

Length

Is the size in bytes of the *KeyValueInformation* buffer, which the caller should set according to the given *KeyValueInformationClass*.

ResultLength

Points to number of bytes actually returned to *KeyValueInformation* or, if the input *Length* is too small, points to the number of bytes required for the available information.

Include

wdm.h or *ntddk.h*

Return Value

ZwQueryValueKey returns STATUS_SUCCESS if it returned the requested information in the *KeyValueInformation* buffer. Otherwise, **ZwQueryValueKey** can return one of the following:

STATUS_BUFFER_TOO_SMALL
STATUS_INVALID_PARAMETER
STATUS_OBJECT_NAME_NOT_FOUND

Comments

The *KeyHandle* passed to **ZwQueryValueKey** must have been opened with the KEY_QUERY_VALUES *DesiredAccess* flag set for this call to succeed. See **ZwCreateKey** for a description of possible values for *DesiredAccess*.

Callers of **ZwQueryValueKey** must be running at IRQL PASSIVE_LEVEL.

See Also

KEY_VALUE_BASIC_INFORMATION, KEY_VALUE_FULL_INFORMATION, KEY_VALUE_PARTIAL_INFORMATION, **ZwCreateKey**, **ZwEnumerateValueKey**, **ZwOpenKey**

ZwReadFile

```
NTSTATUS
  ZwReadFile(
  IN HANDLE  FileHandle,
  IN HANDLE  Event  OPTIONAL,
  IN PIO_APC_ROUTINE  ApcRoutine  OPTIONAL,
  IN PVOID  ApcContext  OPTIONAL,
  OUT PIO_STATUS_BLOCK  IoStatusBlock,
  OUT PVOID  Buffer,
  IN ULONG  Length,
  IN PLARGE_INTEGER  ByteOffset  OPTIONAL,
  IN PULONG  Key  OPTIONAL
  );
```

Data can be read from an opened file using **ZwReadFile**.

Parameters

FileHandle

Specifies the handle returned by a successful call to **ZwCreateFile**.

Event

Specifies an optional handle for an event to be set to the signaled state after the read operation completes. Device and intermediate drivers should set this parameter to NULL.

ApcRoutine

Device and intermediate drivers should set this pointer to NULL.

ApcContext

Device and intermediate drivers should set this pointer to NULL.

IoStatusBlock

Pointer to a variable that receives the final completion status and information about the requested read operation.

Buffer

Pointer to a caller-allocated buffer that receives the data read from the file.

Length

Specifies the size in bytes of the given *Buffer*. A successful call to **ZwReadFile** returns the given number of bytes from the file, unless this routine reaches the end of file first.

ByteOffset

Pointer to a variable that specifies the starting byte offset in the file where the read operation will begin. If an attempt is made to read beyond the end of the file, **ZwReadFile** returns an error.

If the call to **ZwCreateFile** set either of the *CreateOptions* flags FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT, the I/O Manager maintains the current file position. If so, the caller of **ZwReadFile** can specify that the current file position offset be used instead of an explicit *ByteOffset* value. This specification can be made by using one of the following methods:

- Specify the system-defined value FILE_USE_FILE_POINTER_POSITION.
- Pass a NULL pointer for *ByteOffset*.

ZwReadFile updates the current file position by adding the number of bytes read when it completes the read operation, if it is using the current file position maintained by the I/O Manager.

Even when the I/O Manager is maintaining the current file position, the caller can reset this position by passing an explicit *ByteOffset* value to **ZwReadFile**. Doing this automatically changes the current file position to that *ByteOffset* value, performs the read operation, and then updates the position according to the number of bytes actually read. This technique gives the caller atomic seek-and-read service.

Key

Device and intermediate drivers should set this pointer to NULL.

Include

wdm.h or *ntddk.h*

Return Value

ZwReadFile either returns STATUS_SUCCESS or the appropriate error status. The number of bytes actually read from the file is returned in the **Information** member of the *IoStatusBlock*.

Comments

Callers of **ZwReadFile** must have already called **ZwCreateFile** with the *DesiredAccess* flag FILE_READ_DATA set, either explicitly or by setting this flag using GENERIC_READ.

If the preceding call to **ZwCreateFile** set the *CreateOptions* flag FILE_NO_INTERMEDIATE_BUFFERING, certain restrictions on the parameters to **ZwReadFile** are enforced. See **ZwCreateFile** for specifics.

ZwReadFile begins reading from the given *ByteOffset* or the current file position into the given *Buffer*. It terminates the read operation under one of the following conditions:

- The buffer is full because the number of bytes specified by the *Length* parameter has been read. Therefore, no more data can be placed into the buffer without an overflow.
- The end of file is reached during the read operation, so there is no more data in the file to be transferred into the buffer.

If the caller opened the file with the *DesiredAccess* SYNCHRONIZE flag set, the caller can wait for this routine to set the given *FileHandle* to the signaled state.

Drivers should call **ZwReadFile** in the context of the system process in three cases:

1. The driver creates the file handle that it passes to **ZwReadFile**.
2. **ZwReadFile** notifies the driver of I/O completion by means of an event created by the driver.
3. **ZwReadFile** notifies the driver of I/O completion by means of an APC callback routine that the driver passes to **ZwReadFile**.

File and event handles are only valid in the process context where the handles are created. Therefore, to avoid security holes, the driver should create any file or event handle that it passes to **ZwReadFile** in the context of the system process instead of the process context that the driver is in.

Likewise, **ZwReadFile** should be called in the context of the system process if it notifies the driver of I/O completion by means of an APC, because APCs are always fired in the context of the thread issuing the IO request. If the driver calls **ZwReadFile** in the context of a process other than the system process, the APC could be delayed indefinitely, or it might not fire at all.

Callers of **ZwReadFile** must be running at IRQL PASSIVE_LEVEL.

See Also

KeInitializeEvent, **ZwCreateFile**, **ZwQueryInformationFile**, **ZwSetInformationFile**, **ZwWriteFile**

ZwSetInformationFile

```
NTSTATUS
  ZwSetInformationFile(
  IN HANDLE  FileHandle,
  OUT PIO_STATUS_BLOCK  IoStatusBlock,
  IN PVOID  FileInformation,
  IN ULONG  Length,
  IN FILE_INFORMATION_CLASS  FileInformationClass
  );
```

ZwSetInformationFile changes various kinds of information about a given file object.

Parameters

FileHandle

Is the handle returned by a successful call to **ZwCreateFile**.

IoStatusBlock

Points to a variable that receives the final completion status and information about the operation.

FileInformation

Points to a buffer or variable containing the information to be set for the file. The contents of *FileInformation* are defined by the *FileInformationClass* parameter, described later. Setting any member of the structure in this buffer or variable to zero tells **ZwSetInformationFile** to leave the current information about the file for that member unchanged.

Length

Specifies the size in bytes of *FileInformation*, which the caller should set according to the given *FileInformationClass*.

FileInformationClass

Specifies the type of information to be reset for the file. Device and intermediate drivers, can specify any of the following:

FileInformationClass Value	Meaning
FileBasicInformation	Change FILE_BASIC_INFORMATION about the file. The caller must have opened the file with the *DesiredAccess* FILE_WRITE_ATTRIBUTES flag set.
FileDispositionInformation	Usually, sets **DeleteFile** in FILE_DISPOSITION_INFORMATION to TRUE, so the file can be deleted when **ZwClose** is called to release the last open handle for the file object. The caller must have opened the file with the *DesiredAccess* DELETE flag set.

Continued

FileInformationClass Value	Meaning
FilePositionInformation	Change the current FILE_POSITION_INFORMATION for the file. The caller must have opened the file with the *DesiredAccess* FILE_READ_DATA or FILE_WRITE_DATA flag set and with either of the *CreateOptions* FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT.
FileEndOfFileInformation	Change the current FILE_END_OF_FILE_INFORMATION for the file: either truncate or extend the amount of valid data in the file by moving the current end-of-file position. The caller must have opened the file with the *DesiredAccess* FILE_WRITE_DATA flag set.

Include

ntddk.h

Return Value

ZwSetInformationFile returns STATUS_SUCCESS or an appropriate error status. It also returns the number of bytes set on the file in the **Information** member of *IoStatusBlock*.

Comments

ZwSetInformationFile changes information about a file. It ignores any member of a FILE_*XXX*_INFORMATION structure that is not supported by a particular device or file system. For example, the FAT file system does not support file-creation times, so **ZwSetInformationFile** ignores the **CreationTime** member of the FILE_BASIC_INFORMATION structure when it is called to change basic file information for files on a FAT partition.

A caller that sets *FileInformationClass* to **FileDispositionInformation** can pass the *FileHandle* subsequently to **ZwClose** but to no other **Zw..File** routine. On return from **ZwSetInformationFile**, the file has been marked for deletion. It is a programming error to attempt any subsequent operation on the open file except closing it.

If the caller sets *FileInformationClass* to **FilePositionInformation** and the preceding call to **ZwCreateFile** set the *CreateOptions* flag FILE_NO_INTERMEDIATE_BUFFERING, certain restrictions on the input FILE_POSITION_INFORMATION **CurrentByteOffset** are enforced. See **ZwCreateFile** for specifics.

If the caller sets *FileInformationClass* to **FileEndOfFileInformation** and the input FILE_END_OF_FILE_INFORMATION **EndOfFile** value specifies an offset beyond the current end-of-file mark, **ZwSetInformationFile** extends the file and writes pad bytes of zeroes between the old and new end-of-file marks.

Callers of **ZwSetInformationFile** must be running at IRQL PASSIVE_LEVEL.

See Also

FILE_BASIC_INFORMATION, FILE_DISPOSITION_INFORMATION, FILE_END_OF_FILE_INFORMATION, FILE_POSITION_INFORMATION, **ZwCreateFile**, **ZwQueryInformationFile**

ZwSetInformationThread

```
NTSTATUS
  ZwSetInformationThread(
  IN HANDLE  ThreadHandle,
  IN THREADINFOCLASS  ThreadInformationClass,
  IN PVOID  ThreadInformation,
  IN ULONG  ThreadInformationLength
  );
```

ZwSetInformationThread can be called to set the priority of a thread for which the caller has a handle.

Parameters

ThreadHandle

Is the open handle for a thread.

ThreadInformationClass

Is one of the system-defined values **ThreadPriority** or **ThreadBasePriority**.

ThreadInformation

Points to a variable specifying the information to be set. If *ThreadInformationClass* is **ThreadPriority**, this value must be > LOW_PRIORITY and <= HIGH_PRIORITY. If *ThreadInformationClass* is **ThreadBasePriority**, this value must fall within the system's valid base priority range and the original priority class for the given thread: that is, if a thread's priority class is variable, that thread's base priority cannot be reset to a real-time priority value and vice versa.

ThreadInformationLength

Is the size in bytes of *ThreadInformation*, which must be at least **sizeof**(KPRIORITY).

Include

ntddk.h

Return Value

ZwSetInformationThread returns STATUS_SUCCESS or an error status, such as STATUS_INFO_LENGTH_MISMATCH or STATUS_INVALID_PARAMETER.

Comments

ZwSetInformationThread can be called by higher-level drivers to set the priority of a thread for which they have a handle.

The caller must have THREAD_SET_INFORMATION access rights for the given thread in order to call this routine.

Usually, device and intermediate drivers that set up driver-created threads call **KeSetBasePriorityThread** or **KeSetPriorityThread** from their driver-created threads, rather than **ZwSetInformationThread**. However, a driver can call **ZwSetInformationThread** to raise the priority of a driver-created thread before that thread is run.

Callers of **ZwSetInformationThread** must be running at IRQL PASSIVE_LEVEL.

See Also

KeSetBasePriorityThread, **KeSetPriorityThread**, **PsCreateSystemThread**

ZwSetValueKey

```
NTSTATUS
  ZwSetValueKey(
  IN HANDLE  KeyHandle,
  IN PUNICODE_STRING  ValueName,
  IN ULONG  TitleIndex  OPTIONAL,
  IN ULONG  Type,
  IN PVOID  Data,
  IN ULONG  DataSize
  );
```

ZwSetValueKey replaces or creates a value entry for a key in the registry.

Parameters

KeyHandle

Is the handle, returned by a successful call to **ZwCreateKey** or **ZwOpenKey**, of key for which a value entry is to be written in the registry.

ValueName

Points to the name of the value entry for which the data is to be written. This parameter can be a NULL pointer if the value entry has no name. If a name string is specified and the given name is not unique relative to its containing key, the data for an existing value entry is replaced.

TitleIndex

Device and intermediate drivers should set this parameter to zero.

Type

Specifies the type of the data to be written for *ValueName*. System-defined types include the following:

REG_*XXX* Type	**Value**
REG_BINARY	Binary data in any form.
REG_DWORD	A 4-byte numerical value.
REG_DWORD_LITTLE_ ENDIAN	A 4-byte numerical value whose least significant byte is at the lowest address, which is identical to type REG_DWORD.
REG_DWORD_BIG_ ENDIAN	A 4-byte numerical value whose least significant byte is at the highest address.
REG_EXPAND_SZ	A zero-terminated Unicode string, containing unexpanded references to environment variables, such as "%PATH%".
REG_LINK	A Unicode string naming a symbolic link; this type is irrelevant to device and intermediate drivers.
REG_MULTI_SZ	An array of zero-terminated strings, terminated by another zero.
REG_NONE	Data with no particular type.
REG_SZ	A zero-terminated Unicode string.
REG_RESOURCE_LIST	A device driver's list of hardware resources, used by the driver or one of the physical devices it controls, in the **\ResourceMap** tree.
REG_RESOURCE_ REQUIREMENTS_LIST	A device driver's list of possible hardware resources it or one of the physical devices it controls can use, from which the system writes a subset into the **\ResourceMap** tree.
REG_FULL_RESOURCE_ DESCRIPTOR	A list of hardware resources that a physical device is using, detected and written into the **\HardwareDescription** tree by the system.

Device drivers need not, and should not attempt to, call **ZwSetValueKey** directly to write value entries in a subkey of the **\Registry..\ResourceMap** key. Only the system can write value entries to the **\Registry..\HardwareDescription** tree.

Data

Points to a caller-allocated buffer containing the data for the value entry.

DataSize

Specifies the size in bytes of the *Data* buffer. If *Type* is any of the REG_*XXX*_SZ, this value must include the terminating zero(s).

Include

wdm.h or *ntddk.h*

Return Value

ZwSetValueKey can return one of the following:

STATUS_SUCCESS
STATUS_ACCESS_DENIED
STATUS_INVALID_HANDLE

Comments

The *KeyHandle* passed to **ZwSetValueKey** must have been opened with the KEY_SET_VALUE *DesiredAccess* flag set for this call to succeed. See **ZwCreateKey** for a description of possible values for *DesiredAccess*.

If the given key has no existing value entry with a name matching the given *ValueName*, **ZwSetValueKey** creates a new value entry with the given name. If a matching value entry name exists, this routine overwrites the original value entry for the given *ValueName*. Thus, **ZwSetValueKey** preserves a unique name for each value entry of any particular key. While each value entry name must be unique to its containing key, many different keys in the registry can have value entries with the same names.

Callers of **ZwSetValueKey** must be running at IRQL PASSIVE_LEVEL.

See Also

HalAssignSlotResources, **IoAssignResources**, **IoQueryDeviceDescription**, **IoReportResourceUsage**, **ZwClose**, **ZwCreateKey**, **ZwFlushKey**, **ZwOpenKey**

ZwUnmapViewOfSection

```
NTSTATUS
  ZwUnmapViewOfSection(
  IN HANDLE  ProcessHandle,
  IN PVOID  BaseAddress
  );
```

ZwUnmapViewOfSection unmaps a view of a section from the virtual address space of a subject process.

Parameters

ProcessHandle

Specifies an open handle of the process that was passed in a preceding call to **ZwMapViewOfSection**.

BaseAddress

Points to the base virtual address of the view that is to be unmapped. This value can be any virtual address within the view.

Include

wdm.h or *ntddk.h*

Return Value

ZwUnmapViewOfSection can return one of the following:

STATUS_NORMAL
STATUS_INVALID_PARAMETER
STATUS_NO_ACCESS

Comments

The entire view of the section specified by the *BaseAddress* parameter is unmapped from the virtual address space of the specified process.

The virtual address region occupied by the view is no longer reserved and is available to map other views or private pages. If the view was also the last reference to the underlying section, then all committed pages in the section are decommitted and the section is deleted.

Callers of **ZwUnmapViewOfSection** must be running at IRQL PASSIVE_LEVEL.

See Also

ZwMapViewOfSection, **ZwOpenSection**

ZwWriteFile

```
NTSTATUS
  ZwWriteFile(
  IN HANDLE  FileHandle,
  IN HANDLE  Event  OPTIONAL,
  IN PIO_APC_ROUTINE  ApcRoutine  OPTIONAL,
  IN PVOID  ApcContext  OPTIONAL,
  OUT PIO_STATUS_BLOCK  IoStatusBlock,
  IN PVOID  Buffer,
  IN ULONG  Length,
  IN PLARGE_INTEGER  ByteOffset  OPTIONAL,
  IN PULONG  Key  OPTIONAL
  );
```

Data can be written to an open file using **ZwWriteFile**.

Parameters

FileHandle

Specifies the handle returned by a successful call to **ZwCreateFile**.

Event

Specifies an optional handle for an event to be set to the signaled state after the write operation completes. Device and intermediate drivers should set this parameter to NULL.

ApcRoutine

Device and intermediate drivers should set this pointer to NULL.

ApcContext

Device and intermediate drivers should set this pointer to NULL.

IoStatusBlock

Pointer to a variable that receives the final completion status and information about the requested write operation.

Buffer

Pointer to a caller-allocated buffer containing the data to be written to the file.

Length

Specifies the size in bytes of the given *Buffer*. A successful call to **ZwWriteFile** transfers the given number of bytes to the file. If necessary, the length of the file is extended.

ByteOffset

Pointer to a variable that specifies the starting byte offset in the file where the write operation will begin. If a given *Length* and *ByteOffset* specify a write operation past the current end-of-file mark, **ZwWriteFile** automatically extends the file and updates the end-of-file mark; any bytes that are not explicitly written between such old and new end-of-file marks are defined to be zero.

If the call to **ZwCreateFile** set only the *DesiredAccess* flag FILE_APPEND_DATA, *ByteOffset* is ignored. Data in the given *Buffer*, for *Length* bytes, is written starting at the current end of file.

If the call to **ZwCreateFile** set either of the *CreateOptions* flags, FILE_SYNCHRONOUS_IO_ALERT or FILE_SYNCHRONOUS_IO_NONALERT, the I/O Manager maintains the current file position. If so, the caller of **ZwWriteFile** can specify that the current file position offset be used instead of an explicit *ByteOffset* value. This specification can be made by using one of the following methods:

- Specify the system-defined value FILE_USE_FILE_POINTER_POSITION.
- Pass a NULL pointer for *ByteOffset*.

ZwWriteFile updates the current file position by adding the number of bytes written when it completes the write operation, if it is using the current file position maintained by the I/O Manager.

Even when the I/O Manager is maintaining the current file position, the caller can reset this position by passing an explicit *ByteOffset* value to **ZwWriteFile**. Doing this automatically changes the current file position to that *ByteOffset* value, performs the write operation, and then updates the position according to the number of bytes actually written. This technique gives the caller atomic seek-and-write service.

It is also possible to cause a write operation to start at the current end of file by specifying FILE_WRITE_TO_END_OF_FILE for the *ByteOffset* parameter even if the I/O Manager is not maintaining the current file position.

Key

Device and intermediate drivers should set this pointer to NULL.

Include

wdm.h or *ntddk.h*

Return Value

ZwWriteFile either returns STATUS_SUCCESS or an appropriate error status. The number of bytes actually written to the file is returned in the **Information** member of *IoStatusBlock*.

Comments

Callers of **ZwWriteFile** must have already called **ZwCreateFile** with the *DesiredAccess* flags FILE_WRITE_DATA and/or FILE_APPEND_DATA set, either explicitly or by setting these flags with GENERIC_WRITE. Note that having only FILE_APPEND_DATA access to a file does not allow the caller to write anywhere in the file except at the current end-of-file mark, while having FILE_WRITE_DATA access to a file does not preclude the caller from writing to or beyond the end of a file.

If the preceding call to **ZwCreateFile** set the *CreateOptions* flag FILE_NO_INTERMEDIATE_BUFFERING, certain restrictions on the parameters to **ZwWriteFile** are enforced. See **ZwCreateFile** for specifics.

ZwWriteFile begins writing data from the given *Buffer* at the given *ByteOffset* in the file, at the current file position within the file, or at the end-of-file mark. It terminates the write operation when it has written *Length* bytes to the file, extending the length of the file if necessary, and resetting the end-of-file mark.

If the caller opened the file with the *DesiredAccess* SYNCHRONIZE flag set, the caller can wait for this routine to set the given *FileHandle* to the signaled state.

Drivers should call **ZwWriteFile** in the context of the system process in three cases:

1. The driver creates the file handle that it passes to **ZwWriteFile**.
2. **ZwWriteFile** notifies the driver of I/O completion by means of an event created by the driver.
3. **ZwWriteFile** notifies the driver of I/O completion by means of an APC callback routine that the driver passes to **ZwWriteFile**.

File and event handles are only valid in the process context where the handles are created. Therefore, to avoid security holes, the driver should create any file or event handle that it passes to **ZwWriteFile** in the context of the system process instead of the process context that the driver is in.

Likewise, **ZwWriteFile** should be called in the context of the system process if it notifies the driver of I/O completion by means of an APC, because APCs are always fired in the context of the thread issuing the I/O request. If the driver calls **ZwWriteFile** in the context of a process other than the system process, the APC could be delayed indefinitely, or it might not fire at all.

Callers of **ZwWriteFile** must be running at IRQL PASSIVE_LEVEL.

See Also

KeInitializeEvent, **ZwCreateFile**, **ZwQueryInformationFile**, **ZwReadFile**, **ZwSetInformationFile**

CHAPTER 12

System Structures

This chapter describes system structures and objects that are parameters to more than one support routine or standard driver routine. It also describes some bus-type-specific and device-type-specific configuration structures that the system defines for the convenience of driver writers.

Other system-defined structures are described in the context of the support routines in preceding chapters, in particular those structures that are relevant only to a single support routine or pair of support routines.

ANSI_STRING

```
typedef struct _STRING {
  USHORT Length;
  USHORT MaximumLength;
  PCHAR Buffer;
} STRING *PANSI_STRING;
```

The STRING structure defines a counted string used for ANSI strings.

Members

Length

The length in bytes of the string stored in **Buffer**.

MaximumLength

The maximum length in bytes of **Buffer**.

Buffer

Points to a buffer used to contain a string of characters.

Include

wdm.h or *ntddk.h*

Comments

The STRING structure is used to pass ANSI strings.

If the string is NULL terminated, **Length** does not include the trailing NULL.

The **MaximumLength** is used to indicate the length of **Buffer** so that if the string is passed to a conversion routine such as **RtlUnicodeStringToAnsiString** the returned string does not exceed the buffer size.

See Also

OEM_STRING, UNICODE_STRING, **RtlAnsiStringToUnicodeSize**, **RtlAnsiStringToUnicodeString**, **RtlFreeAnsiString**, **RtlInitAnsiString**, **RtlUnicodeStringToAnsiString**

CM_EISA_FUNCTION_INFORMATION

```
typedef struct _CM_EISA_FUNCTION_INFORMATION {
  ULONG CompressedId;
  UCHAR IdSlotFlags1;
  UCHAR IdSlotFlags2;
  UCHAR MinorRevision;
  UCHAR MajorRevision;
  UCHAR Selections[26];
  UCHAR FunctionFlags;
  UCHAR TypeString[80];
  EISA_MEMORY_CONFIGURATION EisaMemory[9];
  EISA_IRQ_CONFIGURATION EisaIrq[7];
  EISA_DMA_CONFIGURATION EisaDma[4];
  EISA_PORT_CONFIGURATION EisaPort[20];
  UCHAR InitializationData[60];
} CM_EISA_FUNCTION_INFORMATION, *PCM_EISA_FUNCTION_INFORMATION;
```

CM_EISA_FUNCTION_INFORMATION defines detailed EISA configuration information returned by **HalGetBusData** for the input *BusDataType* **EisaConfiguration**, or by **HalGetBusDataByOffset** for the input *BusDataType* **EisaConfiguration** and the *Offset* zero, assuming the caller-allocated *Buffer* is of sufficient *Length*.

Members

CompressedId

The EISA compressed identification of the device at this slot. The value is identical to the **CompressedId** member of the CM_EISA_SLOT_INFORMATION structure.

IdSlotFlags1

The EISA slot identification flags.

IdSlotFlags2

The EISA slot identification flags.

MinorRevision

Information supplied by the manufacturer.

MajorRevision

Information supplied by the manufacturer.

Selections[26]

The EISA selections for the device.

FunctionFlags

Indicates which of the members has available information. Callers can use the following system-defined masks to determine whether a particular type of configuration information can be or has been returned by **HalGetBusData** or **HalGetBusDataByOffset**:

EISA_FUNCTION_ENABLED
EISA_FREE_FORM_DATA
EISA_HAS_PORT_INIT_ENTRY
EISA_HAS_PORT_RANGE
EISA_HAS_DMA_ENTRY
EISA_HAS_IRQ_ENTRY
EISA_HAS_MEMORY_ENTRY
EISA_HAS_TYPE_ENTRY
EISA_HAS_INFORMATION

The EISA_HAS_INFORMATION mask is a combination of the following:

EISA_HAS_PORT_RANGE
EISA_HAS_DMA_ENTRY
EISA_HAS_IRQ_ENTRY
EISA_HAS_MEMORY_ENTRY
EISA_HAS_TYPE_ENTRY

TypeString[80]

Specifies the type of device.

EisaMemory[9]

Describes the EISA device memory configuration information, defined as follows:

```
typedef struct _EISA_MEMORY_CONFIGURATION {
    EISA_MEMORY_TYPE ConfigurationByte;
    UCHAR DataSize;
    USHORT AddressLowWord;
    UCHAR AddressHighByte;
    USHORT MemorySize;
} EISA_MEMORY_CONFIGURATION, *PEISA_MEMORY_CONFIGURATION;
```

EisaIrq[7]

Describes the EISA interrupt configuration information, defined as follows:

```
typedef struct _EISA_IRQ_CONFIGURATION {
    EISA_IRQ_DESCRIPTOR ConfigurationByte;
    UCHAR Reserved;
} EISA_IRQ_CONFIGURATION, *PEISA_IRQ_CONFIGURATION;
```

EisaDma[4]

Describes the EISA DMA configuration information, defined as follows:

```
typedef struct _EISA_DMA_CONFIGURATION {
    DMA_CONFIGURATION_BYTE0 ConfigurationByte0;
    DMA_CONFIGURATION_BYTE1 ConfigurationByte1;
} EISA_DMA_CONFIGURATION, *PEISA_DMA_CONFIGURATION;
```

EisaPort[20]

Describes the EISA device port configuration information, defined as follows:

```
typedef struct _EISA_PORT_CONFIGURATION {
    EISA_PORT_DESCRIPTOR Configuration;
    USHORT PortAddress;
} EISA_PORT_CONFIGURATION, *PEISA_PORT_CONFIGURATION;
```

InitializationData[60]

Vendor-supplied, device-specific initialization data, if any.

Include

wdm.h or *ntddk.h*

Comments

The information returned by **HalGetBusData** or **HalGetBusDataByOffset** in CM_EISA_FUNCTION_INFORMATION and/or in the CM_EISA_SLOT_INFORMATION header immediately preceding it is read-only.

See Also

CM_EISA_SLOT_INFORMATION, **HalGetBusData**, **HalGetBusDataByOffset**

CM_EISA_SLOT_INFORMATION

```
typedef struct _CM_EISA_SLOT_INFORMATION {
  UCHAR ReturnCode;
  UCHAR ReturnFlags;
  UCHAR MajorRevision;
  UCHAR MinorRevision;
  USHORT Checksum;
  UCHAR NumberFunctions;
  UCHAR FunctionInformation;
  ULONG CompressedId;
} CM_EISA_SLOT_INFORMATION, *PCM_EISA_SLOT_INFORMATION;
```

CM_EISA_SLOT_INFORMATION defines EISA configuration header information returned by **HalGetBusData** for the input *BusDataType* **EisaConfiguration**, or by **HalGetBusDataByOffset** for the input *BusDataType* **EisaConfiguration** and the *Offset* zero, assuming the caller-allocated *Buffer* is of sufficient *Length*.

Members

ReturnCode

Contains a status code if an error occurs when the EISA BIOS is queried. Possible status codes include the following:

EISA_INVALID_SLOT
EISA_INVALID_FUNCTION
EISA_INVALID_CONFIGURATION
EISA_EMPTY_SLOT
EISA_INVALID_BIOS_CALL

ReturnFlags

The return flags.

MajorRevision

Information supplied by the manufacturer.

MinorRevision

Information supplied by the manufacturer.

Checksum

The checksum value, allowing validation of the configuration data.

NumberFunctions

The number at this slot.

FunctionInformation

Whether there is available CM_EISA_FUNCTION_INFORMATION for this slot.

CompressedId

The EISA compressed identification of the device at this slot. This value is identical to the **CompressedId** member of the CM_EISA_FUNCTION_INFORMATION structure. This member can be read to determine whether the caller should call **HalGetBusData** or **HalGetBusDataByOffset** again with sufficient buffer space to get more detailed CM_EISA_FUNCTION_INFORMATION for a device it supports.

Include

wdm.h or *ntddk.h*

Comments

The information returned by **HalGetBusData** or **HalGetBusDataByOffset** in CM_EISA_SLOT_INFORMATION and in CM_EISA_FUNCTION_INFORMATION immediately following it is read-only.

The driver of an EISA device might call **HalGetBusData** or **HalGetBusDataByOffset** for each slot on each EISA bus in the system, requesting only CM_EISA_SLOT_INFORMATION in order to find the device(s) it supports by examining the returned **CompressedId** values. Then, such a driver could allocate sufficient buffer space to call **HalGetBusData(ByOffset)** again for CM_EISA_SLOT_INFORMATION and CM_EISA_FUNCTION_INFORMATION at slots where its device(s) can be found.

See Also

CM_EISA_FUNCTION_INFORMATION, **HalGetBusData**, **HalGetBusDataByOffset**

CM_FLOPPY_DEVICE_DATA

```
typedef struct _CM_FLOPPY_DEVICE_DATA {
  USHORT Version;
  USHORT Revision;
  CHAR Size[8];
  ULONG MaxDensity;
  ULONG MountDensity;
  //
  // New data fields for version >= 2.0
  //
  UCHAR StepRateHeadUnloadTime;
```

```
  UCHAR HeadLoadTime;
  UCHAR MotorOffTime;
  UCHAR SectorLengthCode;
  UCHAR SectorPerTrack;
  UCHAR ReadWriteGapLength;
  UCHAR DataTransferLength;
  UCHAR FormatGapLength;
  UCHAR FormatFillCharacter;
  UCHAR HeadSettleTime;
  UCHAR MotorSettleTime;
  UCHAR MaximumTrackValue;
  UCHAR DataTransferRate;
} CM_FLOPPY_DEVICE_DATA, *PCM_FLOPPY_DEVICE_DATA;
```

CM_FLOPPY_DEVICE_DATA defines a device-type-specific data record that is stored in the **\\Registry\Machine\Hardware\Description** tree for a floppy controller if the system can collect this information during the boot process.

Members

Version

The version number of this structure.

Revision

The revision of this structure.

Size[8]

The floppy disk density size.

MaxDensity

The maximum density.

MountDensity

The mount density.

StepRateHeadUnloadTime

The step rate head unload time in milliseconds.

HeadLoadTime

The head load time in milliseconds.

MotorOffTime

The motor off time in seconds.

SectorLengthCode

Indicates the sector size as an exponent in the formula ((2**code*) * 128).

SectorPerTrack

The number of sectors per track.

ReadWriteGapLength

The read/write gap length, in bytes.

DataTransferLength

The data transfer length, in bytes, not including the synchronization field.

FormatGapLength

The format gap length, in bytes.

FormatFillCharacter

The format fill character.

HeadSettleTime

The head settle time in milliseconds.

MotorSettleTime

The motor settle time in milliseconds.

MaximumTrackValue

The maximum track number on the media. Track numbers are zero-based values.

DataTransferRate

The value written to the Datarate register before accessing the media.

Include

wdm.h or *ntddk.h*

See Also

IoQueryDeviceDescription, **IoReportResourceUsage**, CM_PARTIAL_RESOURCE_DESCRIPTOR

CM_FULL_RESOURCE_DESCRIPTOR

```
typedef struct _CM_FULL_RESOURCE_DESCRIPTOR {
  INTERFACE_TYPE InterfaceType;
  ULONG BusNumber;
  CM_PARTIAL_RESOURCE_LIST PartialResourceList;
} CM_FULL_RESOURCE_DESCRIPTOR, *PCM_FULL_RESOURCE_DESCRIPTOR;
```

A CM_FULL_RESOURCE_DESCRIPTOR structure specifies a set of system hardware resources of various types, assigned to a device that is connected to a specific bus. This structure is contained within a CM_RESOURCE_LIST structure.

Members

InterfaceType

Specifies the type of bus to which the device is connected. This must be one of the types defined by INTERFACE_TYPE, in *wdm.h* or *ntddk.h*. (Not used by WDM drivers.)

BusNumber

The system-assigned, driver-supplied, zero-based number of the bus to which the device is connected. (Not used by WDM drivers.)

PartialResourceList

A CM_PARTIAL_RESOURCE_LIST structure.

Include

wdm.h or *ntddk.h*

See Also

CM_RESOURCE_LIST, CM_PARTIAL_RESOURCE_LIST

CM_INT13_DRIVE_PARAMETER

```
typedef struct _CM_INT13_DRIVE_PARAMETER {
  USHORT DriveSelect;
  ULONG MaxCylinders;
  USHORT SectorsPerTrack;
  USHORT MaxHeads;
  USHORT NumberDrives;
} CM_INT13_DRIVE_PARAMETER, *PCM_INT13_DRIVE_PARAMETER;
```

CM_INT13_DRIVE_PARAMETER defines a device-type-specific data record that is stored in the **\\Registry\Machine\Hardware\Description** tree for a disk controller if the system can collect this information during the boot process.

Members

DriveSelect

The drive selected value.

MaxCylinders

The maximum number of cylinders.

SectorsPerTrack

The number of sectors per track.

MaxHeads

The maximum number of heads.

NumberDrives

The number of drives.

Include

wdm.h or *ntddk.h*

See Also

IoQueryDeviceDescription, **IoReportResourceUsage**

CM_KEYBOARD_DEVICE_DATA

```
typedef struct _CM_KEYBOARD_DEVICE_DATA {
  USHORT Version;
  USHORT Revision;
  UCHAR Type;
  UCHAR Subtype;
  USHORT KeyboardFlags;
} CM_KEYBOARD_DEVICE_DATA, *PCM_KEYBOARD_DEVICE_DATA;
```

CM_KEYBOARD_DEVICE_DATA defines a device-type-specific data record that is stored in the **\\Registry\Machine\Hardware\Description** tree for a keyboard peripheral if the system can collect this information during the boot process.

Members

Version

The version number of this structure.

Revision

The revision of this structure.

Type

The type of the keyboard.

Subtype

The subtype of the keyboard.

KeyboardFlags

Defined by x86 BIOS INT 16h, function 02 as:

Bit	Defined As
7	Insert on.
6	Caps Lock on.
5	Num Lock on.
4	Scroll Lock on.
3	Alt Key is down.
2	Ctrl Key is down.
1	Left shift key is down.
0	Right shift key is down.

Include

wdm.h or *ntddk.h*

See Also

IoQueryDeviceDescription, **IoReportResourceUsage**, CM_PARTIAL_RESOURCE_DESCRIPTOR

CM_MCA_POS_DATA

```
typedef struct _CM_MCA_POS_DATA {
  USHORT AdapterId;
  UCHAR PosData1;
  UCHAR PosData2;
  UCHAR PosData3;
  UCHAR PosData4;
} CM_MCA_POS_DATA, *PCM_MCA_POS_DATA;
```

CM_MCA_POS_DATA defines IBM-compatible MCA POS configuration information for a slot.

Include

wdm.h or *ntddk.h*

See Also

HalGetBusData, HalGetBusDataByOffset

CM_PARTIAL_RESOURCE_DESCRIPTOR

```
typedef struct _CM_PARTIAL_RESOURCE_DESCRIPTOR {
    UCHAR Type;
    UCHAR ShareDisposition;
    USHORT Flags;
    union {
        struct {
            PHYSICAL_ADDRESS Start;
            ULONG Length;
        } Generic;
        struct {
            PHYSICAL_ADDRESS Start;
            ULONG Length;
        } Port;
        struct {
            ULONG Level;
            ULONG Vector;
            ULONG Affinity;
        } Interrupt;
        struct {
            PHYSICAL_ADDRESS Start;
            ULONG Length;
        } Memory;
        struct {
            ULONG Channel;
            ULONG Port;
            ULONG Reserved1;
        } Dma;
        struct {
            ULONG Data[3];
        } DevicePrivate;
        struct {
            ULONG Start;
            ULONG Length;
            ULONG Reserved;
        } BusNumber;
        struct {
            ULONG DataSize;
            ULONG Reserved1;
```

```
            ULONG Reserved2;
        } DeviceSpecificData;
    } u;
} CM_PARTIAL_RESOURCE_DESCRIPTOR, *PCM_PARTIAL_RESOURCE_DESCRIPTOR;
```

The CM_PARTIAL_RESOURCE_DESCRIPTOR structure specifies one or more system hardware resources, of a single type, assigned to a device. This structure is used to create an array within a CM_PARTIAL_RESOURCE_LIST structure.

Members

Type

Identifies the resource type. The constant value specified for **Type** indicates which structure within the **u** union is valid, as indicated in the following table. (These flags are used within both CM_PARTIAL_RESOURCE_DESCRIPTOR and IO_RESOURCE_DESCRIPTOR structures, except where noted.)

Type Value	u Member Substructure
CmResourceTypePort	**u.Port**
CmResourceTypeInterrupt	**u.Interrupt**
CmResourceTypeMemory	**u.Memory**
CmResourceTypeDma	**u.Dma**
CmResourceTypeDevicePrivate	**u.DevicePrivate**
CmResourceTypeBusNumber	**u.BusNumber**
CmResourceTypeDeviceSpecific	**u.DeviceSpecificData** (Not used within IO_RESOURCE_DESCRIPTOR.)
CmResourceTypePcCardConfig	**u.DevicePrivate**
CmResourceTypeMfCardConfig	**u.DevicePrivate**
CmResourceTypeConfigData	*Reserved for system use.*
CmResourceTypeNonArbitrated	*Not used.*

ShareDisposition

Indicates whether the described resource can be shared. Valid constant values are listed in the following table.

Value	Definition
CmResourceShareDeviceExclusive	The device requires exclusive use of the resource.
CmResourceShareDriverExclusive	The driver requires exclusive use of the resource. (Not supported for WDM drivers.)
CmResourceShareShared	The resource can be shared without restriction.

Flags

Contains bit flags that are specific to the resource type, as indicated in the following table. Flags can be OR'ed together as appropriate.

Resource Type	Flag	Definition
CmResourceTypePort		
	CM_RESOURCE_PORT_MEMORY	The device is accessed in memory address space.
	CM_RESOURCE_PORT_IO	The device is accessed in IO address space.
	CM_RESOURCE_PORT_10_BIT_ DECODE	The device decodes 10 bits of the port address.
	CM_RESOURCE_PORT_12_BIT_ DECODE	The device decodes 12 bits of the port address.
CmResourceTypePort		
	CM_RESOURCE_PORT_16_BIT_ DECODE	The device decodes 16 bits of the port address.
	CM_RESOURCE_PORT_POSITIVE_ DECODE	The device uses "positive decode" instead of "subtractive decode". (In general, PCI devices use positive decode and ISA buses use subtractive decode.)
	CM_RESOURCE_PORT_PASSIVE_ DECODE	The device decodes the port but the driver does not use it.
	CM_RESOURCE_PORT_WINDOW_ DECODE	*Reserved for system use.*
CmResourceTypeInterrupt		
	CM_RESOURCE_INTERRUPT_LEVEL_ SENSITIVE	The IRQ line is level-triggered. (These IRQs are usually shareable.)
	CM_RESOURCE_INTERRUPT_ LATCHED	The IRQ line is edge-triggered.
CmResourceTypeMemory		
	CM_RESOURCE_MEMORY_READ_ WRITE	The memory range is readable and writable.
	CM_RESOURCE_MEMORY_READ_ ONLY	The memory range is read-only.
	CM_RESOURCE_MEMORY_WRITE_ ONLY	The memory range is write-only.
	CM_RESOURCE_MEMORY_ PREFETCHABLE	The memory range is pre-fetchable.

Resource Type	Flag	Definition
	CM_RESOURCE_MEMORY_ COMBINEDWRITE	Combined-write caching is allowed.
	CM_RESOURCE_MEMORY_24	The device uses 24-bit addressing.
	CM_RESOURCE_MEMORY_ CACHEABLE	The memory range is cacheable.
CmResourceTypeDma		
	CM_RESOURCE_DMA_8	8-bit DMA channel
	CM_RESOURCE_DMA_16	16-bit DMA channel
CmResourceTypeDma		
	CM_RESOURCE_DMA_32	32-bit DMA channel
	CM_RESOURCE_DMA_8_AND_16	8-bit and 16-bit DMA channel
	CM_RESOURCE_DMA_BUS_MASTER	The device supports bus master DMA transfers.
	CM_RESOURCE_DMA_TYPE_A	Type A DMA
	CM_RESOURCE_DMA_TYPE_B	Type B DMA
	CM_RESOURCE_DMA_TYPE_F	Type F DMA

u.Generic

Not used.

u.Port

Specifies a range of I/O port addresses, using the following members:

Start

For raw resources: Bus-relative physical address of the lowest of a range of contiguous I/O port addresses allocated to the device.

For translated resources: System physical address of the lowest of a range of contiguous I/O port addresses allocated to the device.

Length

The length, in bytes, of the range of allocated I/O port addresses.

u.Interrupt

Specifies an interrupt vector and level, using the following members:

Level

For raw resources: The device's bus-specific IRQL (if appropriate for the platform and bus).

For translated resources: The DIRQL assigned to the device.

Vector

For raw resources: The device's bus-specific interrupt vector (if appropriate for the platform and bus).

For translated resources: The global system vector assigned to the device.

Affinity

A bit mask value indicating the set of processors the device can interrupt. If the device can interrupt any processor, set this to -1.

u.Memory

Specifies a range of memory addresses, using the following members:

Start

For raw resources: Bus-relative physical address of the lowest of a range of contiguous memory addresses allocated to the device.

For translated resources: System physical address of the lowest of a range of contiguous memory addresses allocated to the device.

Length

The length, in bytes, of the range of allocated memory addresses.

u.Dma

Specifies a DMA setting, using one of the following members:

Channel

The number of the DMA channel on a system DMA controller that the device can use.

Port

The number of the DMA port that an MCA-type device can use.

Reserved1

Not used.

u.DevicePrivate

Reserved for system use.

u.BusNumber

Specifies bus numbers, using the following members:

Start

The lowest-numbered of a range of contiguous buses allocated to the device.

Length

The number of buses allocated to the device.

Reserved

Not used.

u.DeviceSpecificData

Specifies the size of a device-specific, private structure that is appended to the end of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure. If **u.DeviceSpecificData** is used, the CM_PARTIAL_RESOURCE_DESCRIPTOR structure must be the last one in the CM_PARTIAL_RESOURCE_LIST array.

DataSize

The number of bytes appended to the end of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

Reserved1

Not used.

Reserved2

Not used.

Examples of device-specific structures include:

CM_FLOPPY_DEVICE_DATA
CM_KEYBOARD_DEVICE_DATA
CM_SCSI_DEVICE_DATA
CM_SERIAL_DEVICE_DATA

Include

wdm.h or *ntddk.h*

Comments

A CM_PARTIAL_RESOURCE_DESCRIPTOR structure can describe either a raw (bus-relative) resource or a translated (system physical) resource, depending on the routine or IRP with which it is being used (see **See Also**).

See Also

CM_RESOURCE_LIST, CM_FULL_RESOURCE_DESCRIPTOR, CM_PARTIAL_RESOURCE_LIST, CM_FLOPPY_DEVICE_DATA, CM_KEYBOARD_DEVICE_DATA, CM_SCSI_DEVICE_DATA, CM_SERIAL_DEVICE_DATA, **IoConnectInterrupt**, **IoGetDeviceProperty**, **IoReportResourceForDetection**, IO_RESOURCE_DESCRIPTOR, IRP_MN_START_DEVICE

CM_PARTIAL_RESOURCE_LIST

```
typedef struct _CM_PARTIAL_RESOURCE_LIST {
  USHORT Version;
  USHORT Revision;
  ULONG Count;
  CM_PARTIAL_RESOURCE_DESCRIPTOR PartialDescriptors[1];
} CM_PARTIAL_RESOURCE_LIST, *PCM_PARTIAL_RESOURCE_LIST;
```

The CM_PARTIAL_RESOURCE_LIST structure specifies a set of system hardware resources, of various types, assigned to a device. This structure is contained within a CM_FULL_RESOURCE_DESCRIPTOR structure.

Members

Version

The version number of this structure. This value should be 1.

Revision

The revision of this structure. This value should be 1.

Count

The number of elements contained in the **PartialDescriptors** array. For WDM drivers, this value is always 1.

PartialDescriptors

An array of CM_PARTIAL_RESOURCE_DESCRIPTOR structures.

Include

wdm.h or *ntddk.h*

See Also

CM_FULL_RESOURCE_DESCRIPTOR, CM_PARTIAL_RESOURCE_DESCRIPTOR

CM_RESOURCE_LIST

```
typedef struct _CM_RESOURCE_LIST {
  ULONG Count;
  CM_FULL_RESOURCE_DESCRIPTOR List[1];
} CM_RESOURCE_LIST, *PCM_RESOURCE_LIST;
```

The CM_RESOURCE_LIST structure specifies all of the system hardware resources assigned to a device.

Members

Count

The number of elements contained in the **List** array. For WDM drivers, this value is always 1.

List

An array of CM_FULL_RESOURCE_DESCRIPTOR structures.

Include

wdm.h or *ntddk.h*

Comments

The CM_RESOURCE_LIST structure defines the format used to store device resource lists in the registry. For more information about hardware resource allocation, see *Hardware Resources* in the *Plug and Play, Power Management, and Setup Design Guide*.

See Also

CM_RESOURCE_LIST, CM_FULL_RESOURCE_DESCRIPTOR, CM_PARTIAL_RESOURCE_LIST, CM_FLOPPY_DEVICE_DATA, CM_KEYBOARD_DEVICE_DATA, CM_SCSI_DEVICE_DATA, CM_SERIAL_DEVICE_DATA, **IoConnectInterrupt**, **IoGetDeviceProperty**, **IoReportResourceForDetection**, IRP_MN_START_DEVICE

CM_SCSI_DEVICE_DATA

```
typedef struct _CM_SCSI_DEVICE_DATA {
  USHORT Version;
  USHORT Revision;
  UCHAR HostIdentifier;
} CM_SCSI_DEVICE_DATA, *PCM_SCSI_DEVICE_DATA;
```

CM_SCSI_DEVICE_DATA defines a device-type-specific data record that is stored in the **\\Registry\\Machine\\Hardware\\Description** tree for a SCSI HBA if the system can collect this information during the boot process.

Members

Version

The version number of this structure.

Revision

The revision for this structure.

HostIdentifier

The SCSI bus identifier used by the ARC firmware.

Include

wdm.h or *ntddk.h*

See Also

IoQueryDeviceDescription, **IoReportResourceUsage**, CM_PARTIAL_RESOURCE_DESCRIPTOR

CM_SERIAL_DEVICE_DATA

```
typedef struct _CM_SERIAL_DEVICE_DATA {
  USHORT Version;
  USHORT Revision;
  ULONG BaudClock;
} CM_SERIAL_DEVICE_DATA, *PCM_SERIAL_DEVICE_DATA;
```

CM_SERIAL_DEVICE_DATA defines a device-type-specific data record that is stored in the **\\Registry\Machine\Hardware\Description** tree for a serial controller if the system can collect this information during the boot process.

Members

Version

The version number of this structure.

Revision

The revision of this structure.

BaudClock

The clock baud rate, in MHz, at which data is transferred.

Include

wdm.h or *ntddk.h*

See Also

IoQueryDeviceDescription, **IoReportResourceUsage**, CM_PARTIAL_RESOURCE_DESCRIPTOR

CONTROLLER_OBJECT

A controller object represents a hardware adapter or controller with homogenous devices that are the actual targets for I/O requests. A controller object can be used to synchronize a device driver's I/O to the target devices through its hardware adapter/controller.

A controller object is partially opaque. Driver writers must know about a certain field associated with the controller object because their drivers access this field through the controller object pointer returned by **IoCreateController**. The following field in a controller object is accessible to the creating driver.

Accessible Fields

PVOID ControllerExtension

Points to the controller extension. The structure and contents of the controller extension are driver-defined. The size is driver-determined, specified in the driver's call to **IoCreateController**. Usually, drivers maintain common state about I/O operations in the controller extension and device-specific state about I/O for a target device in the corresponding device extension.

Include

ntddk.h

Comments

Most driver routines that process IRPs are given a pointer to the target device object. Consequently, device drivers that use controller objects frequently store the controller object pointer returned by **IoCreateController** in each device extension.

Note that a controller object has no name so it cannot be the target of an I/O request, and higher-level drivers cannot connnect or attach their device objects to a device driver's controller object.

Undocumented fields within a controller object should be considered inaccessible. Drivers with dependencies on object field locations or access to undocumented fields might not remain portable and interoperable with other drivers over time.

See Also

IoCreateController

DEVICE_DESCRIPTION

```
typedef struct _DEVICE_DESCRIPTION {
  ULONG Version;
  BOOLEAN Master;
  BOOLEAN ScatterGather;
  BOOLEAN DemandMode;
  BOOLEAN AutoInitialize;
  BOOLEAN Dma32BitAddresses;
  BOOLEAN IgnoreCount;
  BOOLEAN Reserved1;
  BOOLEAN Dma64BitAddresses;
  ULONG BusNumber;
  ULONG DmaChannel;
  INTERFACE_TYPE  InterfaceType;
  DMA_WIDTH DmaWidth;
  DMA_SPEED DmaSpeed;
  ULONG MaximumLength;
  ULONG DmaPort;
} DEVICE_DESCRIPTION, *PDEVICE_DESCRIPTION;
```

DEVICE_DESCRIPTION describes the attributes of the physical device for which a driver is requesting a DMA object.

Members

Version

Specifies the version of this structure. Must be DEVICE_DESCRIPTION_VERSION or, if the **IgnoreCount** field is TRUE, must be DEVICE_DESCRIPTION_VERSION1.

Master

Indicates whether the device runs as a busmaster adapter (TRUE) or a slave DMA device (FALSE).

ScatterGather

Indicates whether the device supports scatter/gather DMA.

DemandMode

Indicates whether to use the system DMA controller's demand mode. Not used for busmaster DMA.

AutoInitialize

Indicates whether to use the system DMA controller's autoinitialize mode. Not used for busmaster DMA.

Dma32BitAddresses

Specifies the use of 32-bit addresses for DMA operations.

IgnoreCount

Indicates whether to ignore the DMA controller's transfer counter. Set to TRUE if the DMA controller in this platform does not maintain an accurate transfer counter, and therefore requires a workaround. If TRUE, **Version** must be set to DEVICE_DESCRIPTION_VERSION1.

Reserved1

Reserved for system use. Must be FALSE.

Dma64BitAddresses

Specifies the use of 64-bit addresses for DMA operations.

BusNumber

Specifies the system-assigned value for the I/O bus. Not used by WDM drivers.

DmaChannel

Specifies the channel number to which a slave device is attached.

InterfaceType

Specifies the type of I/O bus involved in the DMA operation.

DmaWidth

Specifies the DMA data size for system DMA. Possible values are **Width8Bits**, **Width16Bits**, and **Width32Bits**. Not used for busmaster DMA.

DmaSpeed

Specifies one of the following speeds for system DMA: **Compatible**, **TypeA**, **TypeB**, **TypeC**, or **TypeF**. Not used for busmaster DMA.

MaximumLength

Specifies the maximum number of bytes the device can handle in each DMA operation.

DmaPort

Specifies the Microchannel-type bus port number. This parameter is obsolete, but is retained in the structure for compatibility with legacy drivers.

Include

wdm.h or *ntddk.h*

Comments

Drivers of devices that use DMA to transfer data use this structure to pass device information when requesting a DMA object. A driver should first zero-initialize the structure, then fill in the information for its device.

The **InterfaceType** specifies the bus interface. At present, its value can be one of the following: **Internal**, **Isa**, **Eisa**, or **PCIBus**. Additional types of buses will be supported in future versions of the operating system. The upper bound on the types of buses supported is always **MaximumInterfaceType**.

Setting **Version** to DEVICE_DESCRIPTION_VERSION1 and **IgnoreCount** to TRUE indicates that the current platform's DMA controller cannot be relied on to maintain an accurate transfer counter. In platforms with such a DMA controller, the system ignores the DMA counter but must take extra precautions to maintain data integrity during transfer operations. Using this workaround to compensate for a deficient DMA controller degrades the speed of DMA transfers.

A driver should specify **TypeF** as the **DmaSpeed** value only if the machine's ACPI BIOS supports it.

See Also

IoGetDmaAdapter

DEVICE_OBJECT

A device object represents a logical, virtual, or physical device for which a loaded driver handles I/O requests. Every kernel-mode driver must call **IoCreateDevice** one or more times from its AddDevice routine to create its device object(s).

A device object is partially opaque. Driver writers must know about certain fields and system-defined symbolic constants associated with device objects because their drivers must access these fields through the device object pointer returned by **IoCreateDevice** and passed to most standard driver routines. The following fields in device objects are accessible to drivers.

Accessible Fields

PDRIVER_OBJECT DriverObject

Points to the driver object, representing the driver's loaded image, that was input to the **DriverEntry** and **AddDevice** routines.

PDEVICE_OBJECT NextDevice

Points to the next device object, if any, created by the same driver. The I/O Manager updates this list at each successful call to **IoCreateDevice**. A driver that is being unloaded must

walk the list of its device objects and delete them. A driver that re-creates its device objects dynamically also uses this field.

PIRP CurrentIrp

Points to the current IRP if the driver has a StartIo routine whose entry point was set in the driver object and if the driver is currently processing IRP(s). Otherwise, this field is NULL.

ULONG Flags

Device drivers OR this field in their newly created device objects with one or more of the following system-defined values:

Value	Description
DO_BUFFERED_IO or DO_DIRECT_IO	Higher-level drivers OR the field with the same value as the next-lower driver, except possibly for highest-level drivers.
DO_BUS_ENUMERATED_DEVICE	Bus drivers set this flag in the PDO of each device they enumerate. This flag pertains only to the PDO; it must not be set in an FDO or filter DO. Therefore, higher-level drivers layered over a bus driver must not propagate this value up the device stack.
DO_DEVICE_INITIALIZING	The I/O Manager sets this flag when it creates the device object. A device function or filter driver clears the flag in its AddDevice routine, after attaching the device object to the device stack, establishing the device power state, and ORing the field with one of the power flags (if necessary). The PnP Manager checks that the flag is clear after return from AddDevice.
DO_POWER_INRUSH	Drivers of devices that require inrush current when powering on must set this flag. A driver cannot set both this flag and DO_POWER_PAGABLE.
DO_POWER_PAGABLE	Windows® 2000 drivers that are pageable, are not part of the paging path, and do not require inrush current must set this flag. The system calls such drivers at IRQL PASSIVE_LEVEL. Drivers cannot set both this flag and DO_POWER_INRUSH. All WDM and Windows 98 drivers must set DO_ POWER_PAGABLE.
DO_VERIFY_VOLUME	Removable-media drivers set this flag while processing transfer requests. Such drivers should also check for this flag in the target for a transfer request before transferring any data; see the Kernel-Mode Drivers Design Guide for details.

ULONG Characteristics

Set when a driver calls **IoCreateDevice** with one of the following values, as appropriate: FILE_REMOVABLE_MEDIA, FILE_READ_ONLY_DEVICE, FILE_FLOPPY_DISKETTE, FILE_WRITE_ONCE_MEDIA, FILE_DEVICE_SECURE_OPEN (Windows® 2000 and Windows NT® SP5 only).

PVOID DeviceExtension

Points to the device extension. The structure and contents of the device extension are driver-defined. The size is driver-determined, specified in the driver's call to **IoCreateDevice**. Most driver routines that process IRPs are given a pointer to the device object so the device extension is usually every driver's primary global storage area and frequently a driver's only global storage area for objects, resources, and any state the driver maintains about the I/O requests it handles.

DEVICE_TYPE DeviceType

Set when a driver calls **IoCreateDevice** as appropriate for the type of underlying device. A driver writer can define a new FILE_DEVICE_*XXX* with a value in the customer range 32768 to 65535 if none of the system-defined values describes the type of the new device. For a list of the system-defined values, see the FILE_DEVICE_*XXX* in *Determining Required I/O Support by Device Object Type*.

CCHAR StackSize

Specifies the minimum number of stack locations in IRPs to be sent to this driver. **IoCreateDevice** sets this field to one in newly created device objects; lowest-level drivers can therefore ignore this field. The I/O manager automatically sets the **StackSize** field in a higher-level driver's device object to the appropriate value if the driver calls **IoAttachDevice** or **IoAttachDeviceToDeviceStack**. Only a higher-level driver that chains itself over another driver with **IoGetDeviceObjectPointer** must explicitly set the value of **StackSize** in its own device object(s) to (1 + the **StackSize** value of the next-lower driver's device object).

ULONG AlignmentRequirement

Some higher-level drivers, such as a class driver layered over a corresponding port driver, that call **IoGetDeviceObjectPointer** reset this field in their device objects to the value of the next-lower driver's device object. Other higher-level drivers set this field at the discretion of the driver designer or leave it as set by the I/O Manager. Each device driver sets this field in its newly created device object(s) to the greater of (the alignment requirement of the device –1) or (the initialized value of this field), which can be one of the following system-defined values:

FILE_BYTE_ALIGNMENT
FILE_WORD_ALIGNMENT
FILE_LONG_ALIGNMENT

FILE_QUAD_ALIGNMENT
FILE_OCTA_ALIGNMENT
FILE_32_BYTE_ALIGNMENT
FILE_64_BYTE_ALIGNMENT
FILE_128_BYTE_ALIGNMENT
FILE_512_BYTE_ALIGNMENT

Include

wdm.h or *ntddk.h*

Comments

The **DeviceType** range 0 to 32767 is reserved for use by Microsoft®.

Undocumented fields within a device object should be considered inaccessible. Drivers with dependencies on object field locations or access to undocumented fields might not remain portable and interoperable with other drivers over time.

The system-supplied video port driver sets up the fields of the device objects it creates on behalf of video miniport drivers. For more information about these video drivers, see the *Graphics Drivers Design Guide*.

The system-supplied SCSI port driver sets up the fields of the device objects it creates on behalf of HBA miniport drivers. For more information about these SCSI drivers, see the *Kernel-Mode Drivers Design Guide* and Part 3 of this volume.

The system-supplied NDIS library sets up the fields of the device objects it creates on behalf of netcard drivers. For more information about NDIS drivers, see the *Network Drivers Design Guide*.

See Also

DRIVER_OBJECT, **IoAttachDevice**, **IoAttachDeviceToDeviceStack**, **IoCreateDevice**, **IoDeleteDevice**, **IoGetDeviceObjectPointer**

DMA_ADAPTER

```
typedef struct _DMA_ADAPTER {
  USHORT Version;
  USHORT Size;
  PDMA_OPERATIONS DmaOperations;
  .
  .
} DMA_ADAPTER, *PDMA_ADAPTER;
```

DMA_ADAPTER describes a system-defined interface to a DMA controller for a given device. A driver calls **IoGetDmaAdapter** to obtain this structure.

Members

Version

Specifies the version of this structure.

Size

Specifies the size, in bytes, of this structure.

DmaOperations

Points to a DMA_OPERATIONS structure that contains pointers to DMA adapter functions.

Include

wdm.h or *ntddk.h*

Comments

Drivers for devices that use DMA to transfer data use this structure to obtain the addresses of functions that enable use of a DMA controller.

See Also

IoGetDmaAdapter, DMA_OPERATIONS

DMA_OPERATIONS

```
typedef struct _DMA_OPERATIONS {
  ULONG Size;
  PPUT_DMA_ADAPTER PutDmaAdapter;
  PALLOCATE_COMMON_BUFFER AllocateCommonBuffer;
  PFREE_COMMON_BUFFER FreeCommonBuffer;
  PALLOCATE_ADAPTER_CHANNEL AllocateAdapterChannel;
  PFLUSH_ADAPTER_BUFFERS FlushAdapterBuffers;
  PFREE_ADAPTER_CHANNEL FreeAdapterChannel;
  PFREE_MAP_REGISTERS FreeMapRegisters;
  PMAP_TRANSFER MapTransfer;
  PGET_DMA_ALIGNMENT GetDmaAlignment;
  PREAD_DMA_COUNTER ReadDmaCounter;
  PGET_SCATTER_GATHER_LIST GetScatterGatherList;
  PPUT_SCATTER_GATHER_LIST PutScatterGatherList;
} DMA_OPERATIONS, *PDMA_OPERATIONS ;
```

DMA_OPERATIONS provides a table of pointers to functions that control the operation of a DMA controller.

Members

Size

Specifies the size, in bytes, of the DMA_OPERATIONS structure.

PutDmaAdapter

Points to a system-defined routine to free a DMA_ADAPTER structure. See **PutDmaAdapter** for further information.

AllocateCommonBuffer

Points to a system-defined routine to allocate a physically contiguous DMA buffer. See **AllocateCommonBuffer** for further information.

FreeCommonBuffer

Points to a system-defined routine to free a physically contiguous DMA buffer previously allocated by **AllocateCommonBuffer**. See **FreeCommonBuffer** for further information.

AllocateAdapterChannel

Points to a system-defined routine to allocate a channel for DMA operations. See **AllocateAdapterChannel** for further information.

FlushAdapterBuffers

Points to a system-defined routine to flush data from the system or busmaster adapter's internal cache after a DMA operation. See **FlushAdapterBuffers** for further information.

FreeAdapterChannel

Points to a system-defined routine to free a channel previously allocated for DMA operations by **AllocateAdapterChannel**. See **FreeAdapterChannel** for further information.

FreeMapRegisters

Points to a system-defined routine to free map registers allocated for DMA operations. See **FreeMapRegisters** for further information.

MapTransfer

Points to a system-defined routine to begin a DMA operation. See **MapTransfer** for further information.

GetDmaAlignment

Points to a system-defined routine to obtain the DMA alignment requirements of the controller. See **GetDmaAlignment** for further information.

ReadDmaCounter

Points to a system-defined routine to obtain the current transfer count for a DMA operation. See **ReadDmaCounter** for further information.

GetScatterGatherList

Points to a system-defined routine that allocates map registers and creates a scatter/gather list for DMA. See **GetScatterGatherList** for further information.

PutScatterGatherList

Points to a system-defined routine that frees map registers and a scatter/gather list after a DMA operation is complete. See **PutScatterGatherList** for further information.

Include

wdm.h or *ntddk.h*

Comments

All members of this structure, with the exception of **Size**, are pointers to functions that drivers use to undertake DMA operations for their device. Drivers obtain these pointers by calling **IoGetDmaAdapter**.

See Also

AllocateAdapterChannel, AllocateCommonBuffer, FreeAdapterChannel, FreeCommonBuffer, FreeMapRegisters, FlushAdapterBuffers, GetDmaAlignment, GetScatterGatherList, IoGetDmaAdapter, MapTransfer, PutDmaAdapter, PutScatterGatherList, ReadDmaCounter

DRIVER_OBJECT

Each driver object represents the image of a loaded kernel-mode driver. A pointer to the driver object is an input parameter to a driver's **DriverEntry**, **AddDevice**, and optional Reinitialize routines and to its Unload routine, if any.

A driver object is partially opaque. Driver writers must know about certain fields of a driver object to initialize a driver and to unload it if the driver is unloadable. The following fields in the driver object are accessible to drivers.

Accessible Fields

PDEVICE_OBJECT DeviceObject

Points to the device object(s) created by the driver. This field is automatically updated when the driver calls **IoCreateDevice** successfully. A driver can this field and the **NextDevice**

field of the DEVICE_OBJECT to step through a list of all the device objects that the driver created.

PUNICODE_STRING HardwareDatabase

Points to the **\Registry\Machine\Hardware** path to the hardware configuration information in the registry.

PFAST_IO_DISPATCH FastIoDispatch

Points to a structure defining the driver's fast I/O entry points. This field is used only by FSDs and network transport drivers.

PDRIVER_INITIALIZE DriverInit

Is the entry point for the **DriverEntry** routine, which is set up by the I/O Manager. A **DriverEntry** routine is declared as follows:

```
NTSTATUS
(*PDRIVER_INITIALIZE) (
    IN PDRIVER_OBJECT DriverObject,
    IN PUNICODE_STRING RegistryPath
    );
```

PDRIVER_STARTIO DriverStartIo

Is the entry point for the driver's StartIo routine, if any, which is set by the **DriverEntry** routine when the driver initializes. If a driver has no StartIo routine, this field is NULL. A StartIo routine is declared as follows:

```
VOID
(*PDRIVER_STARTIO) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );
```

PDRIVER_UNLOAD DriverUnload

Is the entry point for the driver's Unload routine, if any, which is set by the **DriverEntry** routine when the driver initializes. If a driver has no Unload routine, this field is NULL. An Unload routine is declared as follows:

```
VOID
(*PDRIVER_UNLOAD) (
    IN PDRIVER_OBJECT DriverObject
    );
```

PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION+1]

Is an array of one or more entry points for the driver's Dispatch routines. Each driver must set at least one Dispatch entry point in this array for the IRP_MJ_*XXX* requests that the

driver handles. Any driver can set as many separate Dispatch entry points as the IRP_MJ_*XXX* codes that the driver handles. Each Dispatch routine is declared as follows:

```
NTSTATUS
(*PDRIVER_DISPATCH) (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    );
```

Include

wdm.h or *ntddk.h*

Comments

Each kernel-mode driver's initialization routine should be named **DriverEntry** so the system will load the driver automatically. If this routine's name is something else, the driver writer must define the name of the initialization routine for the linker; otherwise, the OS loader or I/O Manager cannot find the driver's transfer address. The names of other standard driver routines can be chosen at the discretion of the driver writer.

A driver must set its Dispatch entry point(s) in the driver object that is passed in to the **DriverEntry** routine when the driver is loaded. A device driver must set one or more Dispatch entry points for the IRP_MJ_*XXX* that any driver of the same type of device is required to handle. A higher-level driver must set one or more Dispatch entry points for all the IRP_MJ_*XXX* that it must pass on to the underlying device driver. Otherwise, a driver is not sent IRPs for any IRP_MJ_*XXX* for which it does not set up a Dispatch routine in the driver object. For more information about the set of IRP_MJ_*XXX* that drivers for different types of underlying devices are required to handle, see *IRP Function Codes and IOCTLS*.

The **DriverEntry** routine also sets the driver's StartIo and/or Unload entry points, if any, in the driver object.

The **HardwareDatabase** string can be used by device drivers to get hardware configuration information from the registry when the driver is loaded. A driver is given read-only access to this string.

The *RegistryPath* input to the **DriverEntry** routine points to the **\Registry\Machine\System\CurrentControlSet\Services***DriverName* key, where the value entry of *DriverName* identifies the driver. As for the **HardwareDatabase** in the input driver object, a driver is given read-only access to this string.

Undocumented fields within a driver object should be considered inaccessible. Drivers with dependencies on object field locations or access to undocumented fields might not remain portable and interoperable with other drivers over time.

See Also

IoCreateDevice, IoDeleteDevice

FILE_ALIGNMENT_INFORMATION

```
typedef struct _FILE_ALIGNMENT_INFORMATION {
  ULONG AlignmentRequirement;
} FILE_ALIGNMENT_INFORMATION;
```

Members

AlignmentRequirement

Is the buffer alignment required by the underlying device. For a list of system-defined values, see DEVICE_OBJECT.

Include

wdm.h or *ntddk.h*

See Also

DEVICE_OBJECT, **ZwQueryInformationFile**, **ZwSetInformationFile**

FILE_BASIC_INFORMATION

```
typedef struct FILE_BASIC_INFORMATION {
  LARGE_INTEGER CreationTime;
  LARGE_INTEGER LastAccessTime;
  LARGE_INTEGER LastWriteTime;
  LARGE_INTEGER ChangeTime;
  ULONG FileAttributes;
} FILE_BASIC_INFORMATION, *PFILE_BASIC_INFORMATION;
```

Members

CreationTime

The time that the file was created.

LastAccessTime

The last time that the file was accessed.

LastWriteTime

The last time that the file was written to.

ChangeTime

The last time the file was changed.

FileAttributes

The file attributes, which can be any valid combination of the following:

FILE_ATTRIBUTE_READONLY
FILE_ATTRIBUTE_HIDDEN
FILE_ATTRIBUTE_SYSTEM
FILE_ATTRIBUTE_DIRECTORY
FILE_ATTRIBUTE_ARCHIVE
FILE_ATTRIBUTE_NORMAL
FILE_ATTRIBUTE_TEMPORARY
FILE_ATTRIBUTE_ATOMIC_WRITE
FILE_ATTRIBUTE_XACTION_WRITE
FILE_ATTRIBUTE_COMPRESSED
FILE_ATTRIBUTE_HAS_EMBEDDING

Include

wdm.h or *ntddk.h*

Comments

FILE_ATTRIBUTE_NORMAL can neither be set nor returned in combination with any other attributes. All other **FileAttributes** values override this attribute.

All dates and times are in system-time format. Absolute system time is the number of 100-nanosecond intervals since the start of 1601.

See Also

KeQuerySystemTime, ZwCreateFile, ZwQueryInformationFile, ZwSetInformationFile

FILE_DISPOSITION_INFORMATION

```
typedef struct _FILE_DISPOSITION_INFORMATION {
  BOOLEAN DeleteFile;
} FILE_DISPOSITION_INFORMATION;
```

Members

DeleteFile

If set to TRUE, delete the file when it is closed.

Include

wdm.h or *ntddk.h*

Comments

The caller must have DELETE access to a given file in order to call **ZwSetInformationFile** with **DeleteFile** set to TRUE in this structure. Subsequently, the only legal operation by such a caller is to close the open file handle.

A file marked for deletion is not actually deleted until all open handles for the file object have been closed and the link count for the file is zero.

See Also

ZwClose, **ZwSetInformationFile**

FILE_END_OF_FILE_INFORMATION

```
typedef struct _FILE_END_OF_FILE_INFORMATION {
  LARGE_INTEGER EndOfFile;
} FILE_END_OF_FILE_INFORMATION;
```

Members

EndOfFile

The absolute new end of file position as a byte offset from the start of the file.

Include

wdm.h or *ntddk.h*

Comments

EndOfFile specifies the byte offset to the end of the file. Because this value is zero-based, it actually refers to the first free byte in the file: that is, it is the offset to the byte immediately following the last valid byte in the file.

See Also

ZwQueryInformationFile, **ZwSetInformationFile**

FILE_FS_DEVICE_INFORMATION

```
typedef struct _FILE_FS_DEVICE_INFORMATION {
  DEVICE_TYPE DeviceType;
  ULONG Characteristics;
} FILE_FS_DEVICE_INFORMATION, *PFILE_FS_DEVICE_INFORMATION;
```

FILE_FS_DEVICE_INFORMATION provides file system device information about the type of device object associated with a file object.

Members

DeviceType

Set when a driver calls **IoCreateDevice** as appropriate for the type of underlying device. A driver writer can define a new FILE_DEVICE_*XXX* with a value in the customer range 32768 to 65535 if none of the system-defined values describes the type of a new device. For a list of the system-defined values, see *Determining Required I/O Support by Device Object Type*.

Characteristics

The device characteristics. For a description of relevant values, see DEVICE_OBJECT.

Include

wdm.h or *ntddk.h*

See Also

DEVICE_OBJECT

FILE_FULL_EA_INFORMATION

```
typedef struct _FILE_FULL_EA_INFORMATION {
  ULONG NextEntryOffset;
  UCHAR Flags;
  UCHAR EaNameLength;
  USHORT EaValueLength;
  CHAR EaName[1];
} FILE_FULL_EA_INFORMATION, *PFILE_FULL_EA_INFORMATION;
```

FILE_FULL_EA_INFORMATION provides extended attribute information. This structure is used primarily by network drivers.

Members

NextEntryOffset

The offset of the next FILE_FULL_EA_INFORMATION-type entry. This member is zero if no other entries follow this one.

Flags

Can be zero or can be set with FILE_NEED_EA, indicating that the file to which the EA belongs cannot be interpreted without understanding the associated extended attributes.

EaNameLength

The length in bytes of the **EaName** array. This value does not include a zero-terminator to **EaName**.

EaValueLength

The length in bytes of each EA value in the array.

EaName

An array of characters naming the EA for this entry.

Include

wdm.h or *ntddk.h*

Comments

This structure is longword-aligned. If a set of FILE_FULL_EA_INFORMATION entries is buffered, **NextEntryOffset** value in each entry, except the last, falls on a longword boundary.

The value(s) associated with each entry follows the **EaName** array. That is, an EA's values are located at **EaName** + (**EaNameLength** + 1).

See Also

ZwCreateFile

FILE_NAME_INFORMATION

```
typedef struct _FILE_NAME_INFORMATION {
  ULONG FileNameLength;
  WCHAR FileName[1];
} FILE_NAME_INFORMATION, *PFILE_NAME_INFORMATION;
```

FILE_NAME_INFORMATION provides file system device information about the name of a file object.

Members

FileNameLength

Specifies the length of the file name string.

FileName

Specifies the first character of the file name string. This is followed in memory by the remainder of the string. (See **ZwQueryInformationFile** for details on the syntax of this file name string.)

Include

ntddk.h

See Also

ZwQueryInformationFile

FILE_OBJECT

To user-mode protected subsystems, a file object represents an open instance of a file, device, directory, or volume. To device and intermediate drivers, a file object usually represents a device object.

A file object is partially opaque. Certain types of drivers, such as FSDs and network transport drivers, use some of the fields of file objects. The following fields in file objects are accessible to drivers.

Accessible Fields

PDEVICE_OBJECT DeviceObject

Points to the device object on which the file is opened.

PVOID FsContext

Points to whatever optional state a driver maintains about the file object; otherwise, NULL.

PVOID FsContext2

Points to whatever additional state a driver maintains about the file object; otherwise, NULL.

UNICODE_STRING FileName

Is the name of the file opened on the device, or the **Length** of the string is zero if the device represented by **DeviceObject** is being opened.

Include

wdm.h or *ntddk.h*

Comments

Drivers can use the **FsContext** fields to maintain driver-determined state about an open file object. A driver cannot use these fields of a file object unless it is accessible in the driver's I/O stack location of an IRP.

The remaining fields in a file object are opaque. They are reserved for use by the I/O Manager and file systems.

Undocumented fields within a file object should be considered inaccessible. Drivers with dependencies on object field locations or access to undocumented fields might not remain portable and interoperable with other drivers over time.

A higher-level driver that successfully calls **IoGetDeviceObjectPointer** during initialization is given a pointer to the file object that represents the next-lower driver's device object in user mode. Such a higher-level driver should save the returned file object pointer. To release its reference to this file object, for example when the driver is being unloaded, the driver must call **ObDereferenceObject** with this file object pointer.

See Also

DEVICE_OBJECT, **IoGetDeviceObjectPointer**, **ObDereferenceObject**

FILE_POSITION_INFORMATION

```
typedef struct FILE_POSITION_INFORMATION {
  LARGE_INTEGER CurrentByteOffset;
} FILE_POSITION_INFORMATION, *PFILE_POSITION_INFORMATION;
```

Members

CurrentByteOffset

The byte offset of the current file pointer.

Include

wdm.h or *ntddk.h*

Comments

FILE_READ_DATA or FILE_WRITE_DATA access to the file is required to change this information about the file, and the file must be opened for synchronous I/O.

If the file was opened or created with the FILE_NO_INTERMEDIATE_BUFFERING option, the value of **CurrentByteOffset** must be an integral multiple of the sector size of the underlying device.

See Also

ZwCreateFile, **ZwSetInformationFile**

FILE_STANDARD_INFORMATION

```
typedef struct FILE_STANDARD_INFORMATION {
  LARGE_INTEGER AllocationSize;
  LARGE_INTEGER EndOfFile;
  ULONG NumberOfLinks;

 BOOLEAN DeletePending;
  BOOLEAN Directory;
} FILE_STANDARD_INFORMATION, *PFILE_STANDARD_INFORMATION;
```

Members

AllocationSize

The file allocation size in bytes. Usually, this value is a multiple of the sector or cluster size of the underlying physical device.

EndOfFile

The end of file location as a byte offset.

NumberOfLinks

The number of hard links to the file.

DeletePending

The delete pending status. TRUE indicates that a file deletion has been requested.

Directory

The file directory status. TRUE indicates the file object represents a directory.

Include

wdm.h or *ntddk.h*

Comments

EndOfFile specifies the byte offset to the end of the file. Because this value is zero-based, it actually refers to the first free byte in the file: that is, it is the offset to the byte immediately following the last valid byte in the file.

See Also

ZwCreateFile, **ZwQueryInformationFile**, **ZwSetInformationFile**

IO_RESOURCE_DESCRIPTOR

```
typedef struct _IO_RESOURCE_DESCRIPTOR {
  UCHAR Option;
  UCHAR Type;
  UCHAR ShareDisposition;
  UCHAR Spare1;
  USHORT Flags;
  USHORT Spare2;
  union {
    struct {
      ULONG Length;
      ULONG Alignment;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Port;
    struct {
      ULONG Length;
      ULONG Alignment;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Memory;
    struct {
      ULONG MinimumVector;
      ULONG MaximumVector;
    } Interrupt;
    struct {
      ULONG MinimumChannel;
      ULONG MaximumChannel;
    } Dma;
    struct {
      ULONG Length;
      ULONG Alignment;
      PHYSICAL_ADDRESS MinimumAddress;
      PHYSICAL_ADDRESS MaximumAddress;
    } Generic;
    struct {
      ULONG Data[3];
    } DevicePrivate;
    struct {
      ULONG Length;
      ULONG MinBusNumber;
      ULONG MaxBusNumber;
      ULONG Reserved;
    } BusNumber;
```

```
    struct {
      ULONG Priority;
      ULONG Reserved1;
      ULONG Reserved2;
    } ConfigData;
  } u;
} IO_RESOURCE_DESCRIPTOR, *PIO_RESOURCE_DESCRIPTOR;
```

An IO_RESOURCE_DESCRIPTOR structure describes a range of raw hardware resources, of one type, that can be used by a device. An array of IO_RESOURCE_DESCRIPTOR structures is contained within each IO_RESOURCE_LIST structure.

Members

Option

Specifies whether this resource description is required, preferred, or alternative. One of the following values must be used:

Value	Definition
0	The specified resource range is required, unless alternative ranges are also specified.
IO_RESOURCE_PREFERRED	The specified resource range is preferred to any alternative ranges.
IO_RESOURCE_ALTERNATIVE	The specified resource range is an alternative to the range preceding it. For example, if one IO_RESOURCE_DESCRIPTOR structure specifies IRQ 5, with IO_RESOURCE_PREFERRED set, and the next structure specifies IRQ 3, with IO_RESOURCE_ALTERNATIVE set, the PnP Manager assigns IRQ 3 to the device only if IRQ 5 is unavailable. (Multiple alternatives can be specified for each resource. Both IO_RESOURCE_ALTERNATIVE and IO_RESOURCE_PREFERRED can be set, indicating a preferred alternative.)
IO_RESOURCE_DEFAULT	*Not used.*

Type

Identifies the resource type. For a list of valid values, see the **Type** member of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

ShareDisposition

Indicates whether the described resource can be shared. For a list of valid values, see the **ShareDisposition** member of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

Flags

Contains bit flags that are specific to the resource type. For a list of valid flags, see the **Flags** member of the CM_PARTIAL_RESOURCE_DESCRIPTOR structure.

u.Port

Specifies a range of I/O port addresses, using the following members:

Length

The length, in bytes, of the range of assignable I/O port addresses.

Alignment

The alignment, in bytes, that the assigned starting address must adhere to. The assigned starting address must be devisable by **Alignment**.

MinimumAddress

The minimum bus-relative I/O port address that can be assigned to the device.

MaximumAddress

The maximum bus-relative I/O port address that can be assigned to the device.

u.Memory

Specifies a range of memory addresses, using the following members:

Length

The length, in bytes, of the range of assignable memory addresses.

Alignment

The alignment, in bytes, that the assigned starting address must adhere to. The assigned starting address must be devisable by **Alignment**.

MinimumAddress

The minimum bus-relative memory address that can be assigned to the device.

MaximumAddress

The maximum bus-relative memory address that can be assigned to the device.

u.Interrupt

Specifies an interrupt vector range, using the following members:

MinimumVector

The minimum bus-relative vector that can be assigned to the device.

MaximumVector

The maximum bus-relative vector that can be assigned to the device.

u.Dma

Specifies a DMA setting, using one of the following members:

MinimumChannel

The minimum bus-relative DMA channel that can be assigned to the device.

MaximumChannel

The maximum bus-relative DMA channel that can be assigned to the device.

u.Generic

Not used.

u.DevicePrivate

Reserved for system use.

u.BusNumber

Specifies bus numbers, using the following members:

Length

The number of bus numbers required.

MinBusNumber

The minimum bus-relative bus number that can be assigned to the device.

MaxBusNumber

The maximum bus-relative bus number that can be assigned to the device.

Reserved

Not used.

u.ConfigData

Reserved for system use.

Include

wdm.h or *ntddk.h*

See Also

CM_PARTIAL_RESOURCE_DESCRIPTOR, IO_RESOURCE_LIST, IO_RESOURCE_REQUIREMENTS_LIST, **IoConnectInterrupt**

IO_RESOURCE_LIST

```
typedef struct _IO_RESOURCE_LIST {
  USHORT Version;
  USHORT Revision;
  ULONG Count;
  IO_RESOURCE_DESCRIPTOR Descriptors[1];
} IO_RESOURCE_LIST, *PIO_RESOURCE_LIST;
```

An IO_RESOURCE_LIST structure describes a range of raw hardware resources, of various types, that can be used by a device. The resources specified represent a single, acceptable resource configuration for a device. An array of IO_RESOURCE_LIST structures is contained within each IO_RESOURCE_REQUIREMENTS_LIST structure.

Members

Version

The version number of this structure. This value should be 1.

Revision

The revision of this structure. This value should be 1.

Count

The number of elements in the **Descriptors** array.

Descriptors

An array of IO_RESOURCE_DESCRIPTOR structures.

Include

wdm.h or *ntddk.h*

See Also

IO_RESOURCE_DESCRIPTOR, IO_RESOURCE_REQUIREMENTS_LIST

IO_RESOURCE_REQUIREMENTS_LIST

```
typedef struct _IO_RESOURCE_REQUIREMENTS_LIST {
  ULONG ListSize;
  INTERFACE_TYPE InterfaceType; // unused for WDM
  ULONG BusNumber; // unused for WDM
  ULONG SlotNumber;
  ULONG Reserved[3];
  ULONG AlternativeLists;
  IO_RESOURCE_LIST List[1];
} IO_RESOURCE_REQUIREMENTS_LIST, *PIO_RESOURCE_REQUIREMENTS_LIST;
```

An IO_RESOURCE_REQUIREMENTS_LIST structure describes sets of resource configurations that can be used by a device. Each configuration represents a range of raw resources, of various types, that can be used by a device.

Members

ListSize

The total number of bytes that constitute the IO_RESOURCE_REQUIREMENTS_LIST structure, its IO_RESOURCE_LIST array, and the latter's IO_RESOURCE_DESCRIPTOR array.

InterfaceType

Specifies an interface type. This must be one of the types defined by INTERFACE_TYPE, in *wdm.h* or *ntddk.h*. (Not used by WDM drivers.)

BusNumber

A system-assigned, zero-based bus number. (Not used by WDM drivers.)

SlotNumber

A system slot number. (Not used by WDM drivers.)

Reserved

Not used.

AlternativeLists

The number of elements in the **List** array.

List

An array of IO_RESOURCE_LIST structures.

Include

wdm.h or *ntddk.h*

See Also

IO_RESOURCE_DESCRIPTOR, IO_RESOURCE_LIST, IRP_MN_FILTER_RESOURCE_REQUIREMENTS, IRP_MN_QUERY_RESOURCE_REQUIREMENTS

IO_STACK_LOCATION

```
typedef struct _IO_STACK_LOCATION {
  UCHAR MajorFunction;
  UCHAR MinorFunction;
  UCHAR Flags;
```

```
    UCHAR Control;
    //
    // The following parameters depend on the IRP_MJ_XXX that is set
    // in MajorFunction. This declaration shows examples for IRP_MJ_READ,
    // IRP_MJ_WRITE, and IRP_MJ_DEVICE_CONTROL or, possibly,
    // IRP_MJ_INTERNAL_DEVICE_CONTROL requests, as well as for IRP_MJ_SCSI,
    // which is equivalent to IRP_MJ_INTERNAL_DEVICE_CONTROL.
    // For other IRP_MJ_XXX, see the structure definition.
    //
    union {
      .
      .
      struct {
        ULONG Length;
        ULONG Key;
        LARGE_INTEGER ByteOffset;
      } Read;

      struct {
        ULONG Length;
        ULONG Key;
        LARGE_INTEGER ByteOffset;
      } Write;
      .
      .
      struct {
        ULONG OutputBufferLength;
        ULONG InputBufferLength;
        ULONG IoControlCode;          // IOCTL_XXX
        PVOID Type3InputBuffer;
      } DeviceIoControl;
      .
      .
      struct {
        struct _SCSI_REQUEST_BLOCK *Srb;
      } Scsi;
      .
      .
      } Parameters;
    PDEVICE_OBJECT DeviceObject;
    PFILE_OBJECT FileObject;
    .
    .
} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

Each I/O stack location in a given IRP has some common members and some request-type-specific members. The following summarizes the general structure of every stack location.

Members

MajorFunction

Is the IRP_MJ_*XXX* telling the driver what I/O operation is requested.

MinorFunction

Is a subfunction code for **MajorFunction**. The PnP Manager, the Power Manager, file system drivers, and SCSI class drivers set this member for some requests.

Flags

Is set with request-type-specific values and used almost exclusively by file system drivers. However, removable-media device drivers check whether this member is set with SL_OVERRIDE_VERIFY_VOLUME for read requests to determine whether to continue the read operation even if the device object's **Flags** is set with DO_VERIFY_VOLUME. Intermediate drivers layered over a removable-media device driver must copy this member into the I/O stack location of the next-lower driver in all incoming IRP_MJ_READ requests.

Control

Drivers can check this member to determine whether it is set with SL_PENDING_RETURNED. Drivers have read-only access to this member.

Parameters.Xxx

Depends on the value of **MajorFunction**. For more detailed information about which IRP_MJ_*XXX* different types of drivers must handle, and for the **Parameters.***Xxx* for each IRP_MJ_*XXX*, see *IRP Function Codes and IOCTLs* of this manual.

DeviceObject

Is a pointer to the driver-created device object representing the target physical, logical, or virtual device for which this driver is to handle the IRP.

FileObject

Is a pointer to the file object, if any, associated with **DeviceObject**.

Include

wdm.h or *ntddk.h*

Comments

Every higher-level driver is responsible for setting up the I/O stack location for the next-lower driver in each IRP.

In some cases, a higher-level driver layered over a mass-storage device driver is responsible for splitting up large transfer requests for the underlying device driver. In particular, SCSI

class drivers must check the **Parameters.Read.Length** and **Parameters.Write.Length**, determine whether the size of the requested transfer exceeds the underlying HBA's transfer capabilities, and, if so, split the **Length** of the original request into a sequence of partial transfers to satisfy the original IRP.

A higher-level driver's call to **IoCallDriver** sets up the **DeviceObject** pointer to the next-lower-level driver's target device object in the I/O stack location of the lower driver. The I/O Manager passes each higher-level driver's IoCompletion routine a pointer to its own **DeviceObject** when or if the IoCompletion routine is called on completion of the IRP.

If a higher-level driver allocates IRPs to make requests of its own, its IoCompletion routine is passed a NULL **DeviceObject** pointer if that driver neither allocates a stack location for itself nor sets up the **DeviceObject** pointer in its own stack location of the newly allocated IRP.

See Also

IoCallDriver, **IoGetCurrentIrpStackLocation**, **IoGetNextIrpStackLocation**, **IoSetCompletionRoutine**, **IoSetNextIrpStackLocation**, IO_STATUS_BLOCK, IRP

IO_STATUS_BLOCK

```
typedef struct _IO_STATUS_BLOCK {
  NTSTATUS Status;
  ULONG Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

The I/O status block in the IRP is set to indicate the final status of a given request before a driver calls **IoCompleteRequest** with the IRP.

Members

Status

Is the completion status, either STATUS_SUCCESS if the requested operation was completed successfully or an informational, warning, or error STATUS_*XXX* value.

Information

Is set to a request-dependent value. For example, on successful completion of a transfer request, this is set to the number of bytes transferred. If a transfer request is completed with another STATUS_*XXX*, this member is set to zero. See *Input and Output Parameters for Common I/O Requests* in Chapter 13 of this volume for more request-specific information.

Include

wdm.h or *ntddk.h*

Comments

Unless a driver completes an IRP with an error from its Dispatch routine for that IRP_MJ_*XXX*, the lowest-level driver in the chain frequently sets the I/O status block in an IRP to the values that are returned to the original requestor of the I/O operation.

The IoCompletion routine(s) of higher-level drivers usually check the I/O status block in IRPs completed by lower drivers. By design, the I/O status block in an IRP is the only information passed back from the underlying device driver to all higher-level drivers' IoCompletion routines.

See Also

IO_STACK_LOCATION, **IoCompleteRequest**, **IoSetCompletionRoutine**, IRP

IRP

```
typedef struct _IRP {
  .
  .
  PMDL MdlAddress;
  ULONG Flags;
  union {
    struct _IRP *MasterIrp;
    .
    .
    PVOID SystemBuffer;
  } AssociatedIrp;
  .
  .
  IO_STATUS_BLOCK IoStatus;
  KPROCESSOR_MODE RequestorMode;
  BOOLEAN PendingReturned;
  .
  .
  BOOLEAN Cancel;
  KIRQL CancelIrql;
  .
  .
  PDRIVER_CANCEL CancelRoutine;
  PVOID UserBuffer;
  union {
    struct {
    .
    .
    union {
      KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
      struct {
```

```
        PVOID DriverContext[4];
      };
    };
    .
    .
    PETHREAD Thread;
    .
    .
    LIST_ENTRY ListEntry;
    .
    .
    } Overlay;
  .
  .
  } Tail;
} IRP, *PIRP;
```

In addition to the request-specific parameters in each driver's I/O stack location in an IRP, drivers also can use the following members of the IRP structure for various purposes.

Members

MdlAddress

Points to an MDL describing a user buffer for an IRP_MJ_READ or IRP_MJ_WRITE request if the driver set up its device object(s) for direct I/O. Drivers that handle IRP_MJ_INTERNAL_DEVICE_CONTROL requests also use this field if the I/O control code was defined with METHOD_DIRECT. For more information about IOCTLs, see *IRP Function Codes and IOCTLs*.

Flags

File system drivers use this field, which is read-only for all drivers. Network and, possibly, highest-level device drivers also might read this field, which can be set with one or more of the following system-defined masks:

IRP_NOCACHE
IRP_PAGING_IO
IRP_MOUNT_COMPLETION
IRP_SYNCHRONOUS_API
IRP_ASSOCIATED_IRP
IRP_BUFFERED_IO
IRP_DEALLOCATE_BUFFER
IRP_INPUT_OPERATION
IRP_SYNCHRONOUS_PAGING_IO
IRP_CREATE_OPERATION
IRP_READ_OPERATION

IRP_WRITE_OPERATION
IRP_CLOSE_OPERATION
IRP_DEFER_IO_COMPLETION

AssociatedIrp.MasterIrp

Points to the master IRP in an IRP that was created by a highest-level driver's call to **IoMakeAssociatedIrp**.

AssociatedIrp.SystemBuffer

Points to a system-space buffer for one of the following:

1. A transfer request to a driver that set up its device object(s) requesting buffered I/O
2. An IRP_MJ_DEVICE_CONTROL request
3. An IRP_MJ_INTERNAL_DEVICE_CONTROL request with an I/O control code that was defined with METHOD_BUFFERED

In any case, the underlying device driver usually transfers data to or from this buffer.

IoStatus

Is the I/O status block in which a driver stores status and information before calling **IoCompleteRequest**.

RequestorMode

Indicates the execution mode of the original requestor of the operation, one of **UserMode** or **KernelMode**.

PendingReturned

If set to TRUE, a driver has marked the IRP pending. Each IoCompletion routine should check the value of this flag. If the flag is TRUE, and if the IoCompletion routine will not return STATUS_MORE_PROCESSING_REQUIRED, the routine should call **IoMarkIrpPending** to propagate the pending status to drivers above it in the device stack.

Cancel

If set to TRUE, the IRP either is or should be cancelled.

CancelIrql

Is the IRQL at which a driver is running when **IoAcquireCancelSpinLock** is called.

CancelRoutine

Is the entry point for a driver-supplied Cancel routine to be called if the IRP is cancelled. NULL indicates that the IRP is not currently cancelable.

UserBuffer

Contains the address of an output buffer if the major function code in the I/O stack location is IRP_MJ_INTERNAL_DEVICE_CONTROL and the I/O control code was defined with METHOD_NEITHER.

Tail.Overlay.DeviceQueueEntry

If IRPs are queued in the device queue associated with the driver's device object, this field links IRPs in the device queue. These links can be used only while the driver is processing the IRP.

Tail.Overlay.DriverContext

If IRPs are not queued in the device queue associated with the driver's device object, this field can be used by the driver to store up to four pointers. This field can be used only while the driver owns the IRP.

Tail.Overlay.Thread

Is a pointer to the caller's thread control block. Higher-level drivers that allocate IRPs for lower-level removable-media drivers must set this field in the IRPs they allocate. Otherwise, the FSD cannot determine which thread to notify if the underlying device driver indicates that the media requires verification.

Tail.Overlay.ListEntry

If a driver manages its own internal queue(s) of IRPs, it uses this field to link one IRP to the next. These links can be used only while the driver is holding the IRP in its queue or is processing the IRP.

Include

wdm.h or *ntddk.h*

Comments

Undocumented members of the IRP are reserved, used only by the I/O Manager or, in some cases, by FSDs.

Each IRP also has one or more I/O stack locations for the driver(s) that process the request. A driver must call **IoGetCurrentIrpStackLocation** to get a pointer to its own stack location in each IRP. Higher-level drivers must call **IoGetNextIrpStackLocation** to get a pointer to the next-lower driver's stack location so the higher-level driver can set it up before calling **IoCallDriver** with the IRP.

While a higher-level driver might check the value of the **Cancel** Boolean in an IRP, that driver cannot assume the IRP will be completed with STATUS_CANCELLED by a lower-level driver even if the value is TRUE.

See Also

IoCreateDevice, **IoGetCurrentIrpStackLocation**, **IoGetNextIrpStackLocation**, **IoSetCancelRoutine**, **IoSetNextIrpStackLocation**, IO_STACK_LOCATION, IO_STATUS_BLOCK

KEY_BASIC_INFORMATION

```
typedef struct _KEY_BASIC_INFORMATION {
  LARGE_INTEGER LastWriteTime;
  ULONG   TitleIndex;
  ULONG   NameLength;
  WCHAR   Name[1];            // Variable-length string
} KEY_BASIC_INFORMATION, *PKEY_BASIC_INFORMATION;
```

KEY_BASIC_INFORMATION defines a subset of the full information available for a registry key.

Members

LastWriteTime

The last time the key or any of its values changed.

TitleIndex

Device and intermediate drivers should ignore this member.

NameLength

Specifies the size in bytes of the following name.

Name

A string of Unicode characters naming the key. The string is *not* null-terminated.

Include

wdm.h or *ntddk.h*

See Also

ZwEnumerateKey, **ZwQueryKey**

KEY_FULL_INFORMATION

```
typedef _KEY_FULL_INFORMATION {
  LARGE_INTEGER LastWriteTime;
  ULONG TitleIndex;
  ULONG ClassOffset;
```

```
  ULONG ClassLength;
  ULONG SubKeys;
  ULONG MaxNameLen;
  ULONG MaxClassLen;
  ULONG Values;
  ULONG MaxValueNameLen;
  ULONG MaxValueDataLen;
  WCHAR Class[1];
} KEY_FULL_INFORMATION; PKEY_FULL_INFORMATION
```

KEY_FULL_INFORMATION defines the information available for a registry key, includeing information about its subkeys and the maximum length for their names and value entries. This information can be u641ed to size buffers to get the names of subkeys and their value entries.

Members

LastWriteTime

Specifies the last time the key or any of its values changed.

TitleIndex

Device and intermediate drivers should ignore this member.

ClassOffset

Specifies the offset from the start of this structure to the **Class** member.

ClassLength

Specifies the number of bytes in the **Class** name.

SubKeys

Specifies the number of subkeys for the key.

MaxNameLen

Specifies the maximum length in bytes of any name for a subkey.

MaxClassLen

Specifies the maximum length in bytes for a **Class** name.

Values

Specifies the number of value entries.

MaxValueNameLen

Specifies the maximum length in bytes of any value entry name.

MaxValueDataLen

Specifies the maximum length in bytes of any value entry data field.

Class[1]

A string of Unicode characters naming the class of the key.

Include

wdm.h or *ntddk.h*

See Also

ZwEnumerateKey, ZwQueryKey

KEY_NODE_INFORMATION

```
typedef struct _KEY_NODE_INFORMATION {
  LARGE_INTEGER LastWriteTime;
  ULONG   TitleIndex;
  ULONG   ClassOffset;
  ULONG   ClassLength;
  ULONG   NameLength;
  WCHAR   Name[1];        // Variable-length string
} KEY_NODE_INFORMATION, *PKEY_NODE_INFORMATION;
```

KEY_NODE_INFORMATION defines basic information available for a registry (sub)key.

Members

LastWriteTime

Specifies the last time the key or any of its values changed.

TitleIndex

Device and intermediate drivers should ignore this member.

ClassOffset

Specifies the offset from the start of this structure to the class name string, which is located immediately following the **Name** string.

ClassLength

Specifies the number of bytes in the class name string.

NameLength

Specifies the size in bytes of the following name.

Name

A string of Unicode characters naming the key.

Include

wdm.h or *ntddk.h*

See Also

ZwEnumerateKey, **ZwQueryKey**

KEY_VALUE_BASIC_INFORMATION

```
typedef struct _KEY_VALUE_BASIC_INFORMATION {
  ULONG   TitleIndex;
  ULONG   Type;
  ULONG   NameLength;
  WCHAR   Name[1];            // Variable size
} KEY_VALUE_BASIC_INFORMATION, *PKEY_VALUE_BASIC_INFORMATION;
```

KEY_VALUE_BASIC_INFORMATION defines a subset of the full information available for a value entry of a registry key.

Members

TitleIndex

Device and intermediate drivers should ignore this member.

Type

Specifies the system-defined type for the registry value in the **Data** member, which is one of the following:

REG_*XXX* Type	**Value**
REG_BINARY	Binary data in any form
REG_DWORD	A 4-byte numerical value
REG_DWORD_LITTLE_ ENDIAN	A 4-byte numerical value whose least significant byte is at the lowest address
REG_DWORD_BIG_ENDIAN	A 4-byte numerical value whose least significant byte is at the highest address
REG_EXPAND_SZ	A zero-terminated Unicode string, containing unexpanded references to environment variables, such as "%PATH%"
REG_LINK	A Unicode string naming a symbolic link. This type is irrelevant to device and intermediate drivers

Continued

REG_*XXX* Type	Value
REG_MULTI_SZ	An array of zero-terminated strings, terminated by another zero
REG_NONE	Data with no particular type
REG_SZ	A zero-terminated Unicode string
REG_RESOURCE_LIST	A device driver's list of hardware resources, used by the driver or one of the physical devices it controls, in the **\ResourceMap** tree
REG_RESOURCE_REQUIREMENTS_LIST	A device driver's list of possible hardware resources it or one of the physical devices it controls can use, from which the system writes a subset into the **\ResourceMap** tree
REG_FULL_RESOURCE_DESCRIPTOR	A list of hardware resources that a physical device is using, detected and written into the **\HardwareDescription** tree by the system

NameLength

Specifies the size in bytes of the following value entry name.

Name

A string of Unicode characters naming a value entry of the key.

Include

wdm.h or *ntddk.h*

See Also

ZwEnumerateValueKey, **ZwQueryValueKey**

KEY_VALUE_FULL_INFORMATION

```
typedef struct _KEY_VALUE_FULL_INFORMATION {
  ULONG   TitleIndex;
  ULONG   Type;
  ULONG   DataOffset;
  ULONG   DataLength;
  ULONG   NameLength;
  WCHAR   Name[1];            // Variable size
} KEY_VALUE_FULL_INFORMATION, *PKEY_VALUE_FULL_INFORMATION;
```

KEY_VALUE_FULL_INFORMATION defines information available for a value entry of a registry key.

Members

TitleIndex

Device and intermediate drivers should ignore this member.

Type

Specifies the system-defined type for the registry value(s) following the **Name** member. For a summary of these types, see KEY_VALUE_BASIC_INFORMATION.

DataOffset

Specifies the offset from the start of this structure to the data immediately following the **Name** string.

DataLength

Specifies the number of bytes of registry information for the value entry identified by **Name**.

NameLength

Specifies the size in bytes of the following value entry name.

Name

A string of Unicode characters naming a value entry of the key.

Include

wdm.h or *ntddk.h*

See Also

KEY_VALUE_BASIC_INFORMATION, **ZwEnumerateValueKey**, **ZwQueryValueKey**

KEY_VALUE_PARTIAL_INFORMATION

```
typedef struct _KEY_VALUE_PARTIAL_INFORMATION {
  ULONG   TitleIndex;
  ULONG   Type;
  ULONG   DataLength;
  UCHAR   Data[1];            // Variable size
} KEY_VALUE_PARTIAL_INFORMATION, *PKEY_VALUE_PARTIAL_INFORMATION;
```

KEY_VALUE_PARTIAL_INFORMATION defines a subset of the value information available for a value entry of a registry key.

Members

TitleIndex

Device and intermediate drivers should ignore this member.

Type

Specifies the system-defined type for the registry value in the **Data** member. For a summary of these types, see KEY_VALUE_BASIC_INFORMATION.

DataLength

The size in bytes of the **Data** member.

Data

A value entry of the key.

Include

wdm.h or *ntddk.h*

See Also

KEY_VALUE_BASIC_INFORMATION, **ZwEnumerateValueKey**, **ZwQueryValueKey**

OEM_STRING

```
typedef struct _STRING {
  USHORT Length;
  USHORT MaximumLength;
  PCHAR Buffer;
} STRING *POEM_STRING;
```

The STRING structure defines a counted string used for OEM strings.

Members

Length

Specifies the length in bytes of the string stored in **Buffer**.

MaximumLength

Specifies the maximum length in bytes of **Buffer.**

Buffer

Points to a buffer used to contain a string of characters.

Include

ntddk.h

Comments

The STRING structure is used to pass OEM strings.

If the string is NULL terminated, **Length** does not include the trailing NULL.

See Also

ANSI_STRING, UNICODE_STRING, **RtlAnsiStringToUnicodeSize**, **RtlAnsiStringToUnicodeString**, **RtlFreeAnsiString**, **RtlInitAnsiString**, **RtlUnicodeStringToAnsiString**

PCI_COMMON_CONFIG

```
typedef struct _PCI_COMMON_CONFIG {
  USHORT VendorID;
  USHORT DeviceID;
  USHORT Command;
  USHORT Status;
  UCHAR RevisionID;
  UCHAR ProgIf;
  UCHAR SubClass;
  UCHAR BaseClass;
  UCHAR CacheLineSize;
  UCHAR LatencyTimer;
  UCHAR HeaderType;
  UCHAR BIST;
  union {
     struct _PCI_HEADER_TYPE_0 {
       ULONG BaseAddresses[PCI_TYPE0_ADDRESSES];
       ULONG Reserved1[2];
       ULONG ROMBaseAddress;
       ULONG Reserved2[2];
       UCHAR InterruptLine;
       UCHAR InterruptPin;
       UCHAR MinimumGrant;
       UCHAR MaximumLatency;
     } type0;
   } u;
   UCHAR DeviceSpecific[192];
} PCI_COMMON_CONFIG, *PPCI_COMMON_CONFIG;
```

PCI_COMMON_CONFIG defines standard PCI configuration information returned by **HalGetBusData** or **HalGetBusDataByOffset** for the input *BusDataType* **PCIConfiguration**, assuming the caller-allocated *Buffer* is of sufficient *Length*.

Members

VendorID

Identifies the manufacturer of the device. This must be a value allocated by the PCI SIG.

DeviceID

Identifies the particular device. This value is assigned by the manufacturer.

Command

Accesses the PCI device's control register. Writing a zero to this register renders the device logically disconnected from the PCI bus except for configuration access. Otherwise, the functionality of the register is device-dependent. Possible system-defined bit encodings for this member include:

```
PCI_ENABLE_IO_SPACE
PCI_ENABLE_MEMORY_SPACE
PCI_ENABLE_BUS_MASTER
PCI_ENABLE_SPECIAL_CYCLES
PCI_ENABLE_WRITE_AND_VALIDATE
PCI_ENABLE_VGA_COMPATIBLE_PALETTE
PCI_ENABLE_PARITY
PCI_ENABLE_WAIT_CYCLE
PCI_ENABLE_SERR
PCI_ENABLE_FAST_BACK_TO_BACK
```

Status

Accesses the PCI device's status register. The functionality of this register is device-dependent. Possible system-defined bit encodings for this member include:

```
PCI_STATUS_FAST_BACK_TO_BACK          // read-only
PCI_STATUS_DATA_PARITY_DETECTED
PCI_STATUS_DEVSEL                     // 2 bits wide
PCI_STATUS_SIGNALED_TARGET_ABORT
PCI_STATUS_RECEIVED_TARGET_ABORT
PCI_STATUS_RECEIVED_MASTER_ABORT
PCI_STATUS_SIGNALED_SYSTEM_ERROR
PCI_STATUS_DETECTED_PARITY_ERROR
```

RevisionID

Specifies the revision level of the device described by the **DeviceID** member. This value is assigned by the manufacturer.

ProgIf

Identifies the register-level programming interface, if any, for the device, according to the PCI classification scheme.

SubClass

Identifies the subtype, if any, of the device, according to the PCI classification scheme.

BaseClass

Identifies type of the device, according to the PCI classification scheme.

CacheLineSize

Contains the system cache line size in 32-bit units. This member is relevant only for PCI busmaster devices. The system determines this value during the boot process.

LatencyTimer

Contains the value of the latency timer in units of PCI bus clocks. This member is relevant only for PCI busmaster devices. The system determines this value during the boot process.

HeaderType

The system ORs the value of this member with PCI_MULTIFUNCTION, if appropriate to the device. The value of this member indicates the PCI_HEADER_TYPE_0 layout that follows.

BIST

Zero indicates that the device does not support built-in self test. Otherwise, the device supports built-in self test according to the PCI standard.

u.type0

Drivers call **HalAssignSlotResources** to configure these values and to get back the bus-relative values passed to other configuration routines.

DeviceSpecific

Contains any device-specific initialization information that is available.

Include

wdm.h or *ntddk.h*

Comments

Certain members of this structure have read-only values, so attempts to reset them are ignored. These members include the following: **VendorID**, **DeviceID**, **RevisionID**, **ProgIf**, **SubClass**, **BaseClass**, **HeaderType**, **InterruptPin**, **MinimumGrant**, and **Maximum-Latency**.

Other members are provisionally read-only: that is, the system initializes them to their correct values, so drivers can safely treat them as read-only. However, they can be reset if a busmaster driver finds it necessary. These members include the following: **CacheLineSize** and **LatencyTimer**.

See Also

HalAssignSlotResources, HalGetBusData, HalGetBusDataByOffset, HalSetBusData, HalSetBusDataByOffset

PCI_SLOT_NUMBER

```
typedef struct _PCI_SLOT_NUMBER {
  union {
    struct {
      ULONG DeviceNumber:5;
      ULONG FunctionNumber:3;
      ULONG Reserved:24;
    } bits;
    ULONG AsULONG;
  } u;
} PCI_SLOT_NUMBER, *PPCI_SLOT_NUMBER;
```

PCI_SLOT_NUMBER defines the format of the *Slot* parameter to the **Hal..BusData** routines when they are called with the *BusDataType* value **PCIConfiguration**.

Members

u.bits

Specifies the particular device on a multifunction adapter at the given slot that is being configured. The **DeviceNumber** indicates the logical slot number for the adapter; the **FunctionNumber** indicates the particular device on that adapter.

u.AsULONG

Species the logical slot number of the device being configured.

Include

wdm.h or *ntddk.h*

Comments

Drivers of PCI devices can call **HalGetBusData** or **HalGetBusDataByOffset** more than once for the same slot number to get the configuration information for their device(s).

For example, a driver might search for devices it supports on all PCI buses in the machine first, and then call **HalGetBusData(ByOffset)** again to request more configuration information about devices of interest. Such a driver could code a loop that calls **HalGetBusData(ByOffset)** with an input *Buffer* of sufficient *Length* only to contain enough of the PCI_COMMON_CONFIG to determine the **VendorID** and **DeviceID** of each PCI device. After finding the *Slot* numbers for any promising PCI devices, the driver would call

HalGetBusData or **HalGetBusDataByOffset** one or more times with additional buffer space to get the information needed to configure its device(s).

See Also

HalAssignSlotResources, **HalGetBusData**, **HalGetBusDataByOffset**, **HalSetBusData**, **HalSetBusDataByOffset**, PCI_COMMON_CONFIG

POOL_TYPE

Specifies the type of system memory to allocate. May be one of the following values:

PagedPool

Paged pool, which is pageable system memory. Paged pool can only be allocated and accessed at IRQL <= DISPATCH_LEVEL.

PagedPoolCacheAligned

Paged pool, aligned on processor cache boundaries.

NonPagedPool

Non-paged pool, which is non-pageable system memory. Non-paged pool can be accessed from any IRQL, but it is a scarce resource and drivers should allocate it only when necessary.

The system can only allocate buffers larger than PAGE_SIZE from non-paged pool in multiples of PAGE_SIZE. Requests for buffers larger than PAGE_SIZE. but not a PAGE_SIZE multiple, waste non-pageable memory.

NonPagedPoolMustSucceed

Non-paged pool that the system reserves for emergency allocations. Drivers must only allocate memory from this pool to avert a system crash.

NonPagedPoolCacheAligned

Non-paged pool, aligned on processor cache boundaries.

NonPagedPoolCacheAlignedMustS

The cache-aligned equivalent of **NonPagedPoolMustSucceed**.

When the system allocates a buffer from pool memory bigger than PAGE_SIZE, it aligns the buffer on a page boundary. Memory requests smaller than (or equal to) PAGE_SIZE are not necessarily aligned on page boundaries, but always fit within a single page, and are aligned on an 8-byte boundary.

See Also

ExAllocatePool, ExAllocatePoolWithTag, ExAllocatePoolWithQuota, ExAllocatePoolWithQuotaTag, ExAllocatePoolWithTagPriority

RTL_OSVERSIONINFOW

```
typedef struct _OSVERSIONINFOW {
  ULONG dwOSVersionInfoSize;
  ULONG dwMajorVersion;
  ULONG dwMinorVersion;
  ULONG dwBuildNumber;
  ULONG dwPlatformId;
  WCHAR  szCSDVersion[ 128 ];     // Maintenance string for PSS usage
} RTL_OSVERSIONINFOW;
```

The RTL_OSVERSIONINFOW structure contains operating system version information. The information includes major and minor version numbers, a build number, a platform identifier, and descriptive text about the operating system. The RTL_OSVERSIONINFOW structure is used with **RtlGetVersion**.

Members

dwOSVersionInfoSize

Specifies the size in bytes of an RTL_OSVERSIONINFOW structure. This member must be set before the structure is used with **RtlGetVersion**.

dwMajorVersion

Identifies the major version number of the operating system. For example, for Windows NT 4, the major version number is four and for Windows 2000 the major version number is five.

dwMinorVersion

Identifies the minor version number of the operating system. For example, for Windows 2000, the minor version number is zero.

dwBuildNumber

Identifies the build number of the operating system.

dwPlatformId

Identifies the operating system platform. For Microsoft® Win32® on Windows NT/ Windows 2000, **RtlGetVersion** returns the value VER_PLATFORM_WIN32_NT.

szCSDVersion

Contains a null-terminated string, such as "Service Pack 3", which indicates the latest Service Pack installed on the system. If no Service Pack has been installed, the string is empty.

Include

ntddk.h

See Also

RTL_OSVERSIONINFOEXW, **RtlGetVersion**, **RtlVerifyVersionInfo**

RTL_OSVERSIONINFOEXW

```
typedef struct _OSVERSIONINFOEXW {
  ULONG dwOSVersionInfoSize;
  ULONG dwMajorVersion;
  ULONG dwMinorVersion;
  ULONG dwBuildNumber;
  ULONG dwPlatformId;
  WCHAR  szCSDVersion[ 128 ];     // Maintenance string for PSS usage
  USHORT wServicePackMajor;
  USHORT wServicePackMinor;
  USHORT wSuiteMask;
  UCHAR wProductType;
  UCHAR wReserved;
} RTL_OSVERSIONINFOEXW;
```

The RTL_OSVERSIONINFOEXW structure contains operating system version information. The information includes major and minor version numbers, a build number, a platform identifier, and information about product suites and the latest Service Pack installed on the system. This structure is used with **RtlGetVersion** and **RtlVerifyVersionInfo**.

Members

dwOSVersionInfoSize

Specifies the size in bytes of an RTL_OSVERSIONINFOEXW structure. This member must be set before the structure is used with **RtlGetVersion**.

dwMajorVersion

Identifies the major version number of the operating system. For example, for Windows 2000, the major version number is five.

dwMinorVersion

Identifies the minor version number of the operating system. For example, for Windows 2000, the minor version number is zero.

dwBuildNumber

Identifies the build number of the operating system.

dwPlatformId

Identifies the operating system platform. For Win32 on Windows NT/Windows 2000, **RtlGetVersion** returns the value VER_PLATFORM_WIN32_NT.

szCSDVersion

Contains a null-terminated string, such as "Service Pack 3", which indicates the latest Service Pack installed on the system. If no Service Pack has been installed, the string is empty.

wServicePackMajor

Identifies the major version number of the latest Service Pack installed on the system. For example, for Service Pack 3, the major version number is three. If no Service Pack has been installed, the value is zero.

wServicePackMinor

Identifies the minor version number of the latest Service Pack installed on the system. For example, for Service Pack 3, the minor version number is zero.

wSuiteMask

Identifies the product suites available on the system. This member is a logical OR of zero or more of the following values:

Value	Meaning
VER_SUITE_BACKOFFICE	Microsoft BackOffice® components are installed.
VER_SUITE_DATACENTER	Windows 2000 Datacenter Server is installed.
VER_SUITE_ENTERPRISE	Windows 2000 Advanced Server is installed.
VER_SUITE_SMALLBUSINESS	Microsoft Small Business Server is installed.
VER_SUITE_SMALLBUSINESS_RESTRICTED	Microsoft Small Business Server is installed with the restrictive client license in force.
VER_SUITE_TERMINAL	Terminal Services are installed.

wProductType

Indicates additional information about the system. This member can be one of the following values:

Value	Meaning
VER_NT_WORKSTATION	Windows 2000 Professional
VER_NT_DOMAIN_CONTROLLER	Windows 2000 domain controller
VER_NT_SERVER	Windows 2000 Server

wReserved

Reserved for future use.

Include

ntddk.h

See Also

RTL_OSVERSIONINFOW, **RtlGetVersion**, **RtlVerifyVersionInfo**

SCATTER_GATHER_LIST

```
typedef struct _SCATTER_GATHER_LIST {
  ULONG NumberOfElements;
  ULONG_PTR Reserved;
  SCATTER_GATHER_ELEMENT Elements[];
} SCATTER_GATHER_LIST, *PSCATTER_GATHER_LIST;
```

The SCATTER_GATHER_LIST structure describes a scatter/gather list for DMA.

Members

NumberOfElements

Specifies the number of elements in the *Elements* array.

Reserved

Reserved for future use.

Elements[ ]

Specifies an array of scatter/gather elements that comprise a scatter/gather list. Each element is of type SCATTER_GATHER_ELEMENT, defined as follows:

```
typedef struct _SCATTER_GATHER_ELEMENT {
    PHYSICAL_ADDRESS Address;
    ULONG Length;
    ULONG_PTR Reserved;
} SCATTER_GATHER_ELEMENT, *PSCATTER_GATHER_ELEMENT;
```

Include

wdm.h or *ntddk.h*

Comments

For drivers that perform scatter/gather I/O, the **GetScatterGatherList** routine creates a SCATTER_GATHER_LIST structure and passes this structure to the driver's Adapter-Control routine.

Each entry in the *Elements* array consists of the length of a physically contiguous scatter/gather region and its starting physical address.

See Also

GetScatterGatherList, **PutScatterGatherList**

UNICODE_STRING

```
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR Buffer;
} UNICODE_STRING *PUNICODE_STRING;
```

The UNICODE_STRING structure is used to define UNICODE strings.

Members

Length

The length in bytes of the string stored in **Buffer**.

MaximumLength

The maximum length in bytes of **Buffer**.

Buffer

Points to a buffer used to contain a string of wide characters.

Include

wdm.h or *ntddk.h*

Comments

The STRING structure is used to pass UNICODE strings.

If the string is NULL terminated, **Length** does not include the trailing NULL.

The **MaximumLength** is used to indicate the length of **Buffer** so that if the string is passed to a conversion routine such as **RtlUnicodeStringToAnsiString** the returned string does not exceed the buffer size.

See Also

OEM_STRING, ANSI_STRING, **RtlAnsiStringToUnicodeSize**, **RtlAnsiStringTo-UnicodeString**, **RtlFreeAnsiString**, **RtlInitAnsiString**, **RtlUnicodeStringToAnsiString**

CHAPTER 13

IRP Function Codes and IOCTLs

Each driver-specific I/O stack location in every IRP has a major function code (IRP_MJ_*XXX*) that tells the driver what operation it or the underlying device driver should carry out to satisfy the I/O request. Each kernel-mode driver must set up one or more Dispatch entry points for a required subset of system-defined major function codes that are set in the I/O stack location(s) of IRPs.

The subset of major function codes that a driver must handle depends on the nature of its device. That is, the IRP_MJ_*XXX* requests sent to a keyboard driver are necessarily somewhat different from those sent to a disk driver.

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 entry point 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, which is valid but unsupported by the new intermediate driver, down to the underlying device driver.

File system drivers also handle a required subset of system-defined IRP_MJ_*XXX*, some with subordinate IRP_MN_*XXX*.

This chapter summarizes the basic I/O requests sent to the system device and intermediate drivers so that driver designers can determine which major function codes their new drivers must handle. For the most common IRP_MJ_*XXX* handled by device and intermediate drivers, this chapter summarizes general information about each request. It also discusses the definition of I/O control codes for device-type-specific, device-specific, and driver-specific device I/O control requests.

Determining Required I/O Support by Device Object Type

Every kernel-mode driver must set an appropriate value in the **Type** field of its device objects when they are created. This value determines which IRP_MJ_*XXX* a device or intermediate driver must handle.

The system device and intermediate drivers set one of the following system-defined constants in the **Type** fields of their respective device objects:

```
FILE_DEVICE_BEEP
FILE_DEVICE_CD_ROM
FILE_DEVICE_CONTROLLER
FILE_DEVICE_DISK
FILE_DEVICE_INPORT_PORT
FILE_DEVICE_KEYBOARD
FILE_DEVICE_MIDI_IN
FILE_DEVICE_MIDI_OUT
FILE_DEVICE_MOUSE
FILE_DEVICE_NULL
FILE_DEVICE_PARALLEL_PORT
FILE_DEVICE_PRINTER
FILE_DEVICE_SCANNER
FILE_DEVICE_SERIAL_MOUSE_PORT
FILE_DEVICE_SERIAL_PORT
FILE_DEVICE_SCREEN
FILE_DEVICE_SOUND
FILE_DEVICE_TAPE
FILE_DEVICE_UNKNOWN
FILE_DEVICE_VIDEO
FILE_DEVICE_VIRTUAL_DISK
FILE_DEVICE_WAVE_IN
FILE_DEVICE_WAVE_OUT
FILE_DEVICE_8042_PORT
FILE_DEVICE_MASS_STORAGE
FILE_DEVICE_KS
FILE_DEVICE_CHANGER
FILE_DEVICE_DVD
FILE_DEVICE_BATTERY
```

The FILE_DEVICE_*Xxx* constants are defined in *ntddk.h* and *wdm.h*.

The FILE_DEVICE_DISK specification covers both floppy and fixed-disk devices, as well as disk partitions.

Intermediate drivers usually set the **Type** fields of their respective device objects to that of the underlying device. For example, the system supplied fault-tolerant disk driver, *ftdisk*, has device objects of type FILE_DEVICE_DISK; it does not define new FILE_DEVICE_*XXX* values for the mirror sets, stripe sets, and volume sets it manages.

File system and network drivers set other system-defined FILE_DEVICE_*XXX* in the **Type** fields of their respective device objects.

FILE_DEVICE_*XXX* values in the range 0–32767 are reserved to Microsoft®. All driver writers must use one of these system-defined constants for new drivers when the underlying device corresponds to a type in the preceding list.

However, a driver designer can define another FILE_DEVICE_*XXX* for a new kind of device. Values in the range 32768–65535 are available for Microsoft customers who develop new kinds of kernel-mode drivers.

Input and Output Parameters for Common I/O Requests

Drivers handle IRPs set with some or all of the following major function codes:

```
IRP_MJ_CREATE
IRP_MJ_CLEANUP
IRP_MJ_CLOSE
IRP_MJ_PNP
IRP_MJ_POWER
IRP_MJ_READ
IRP_MJ_WRITE
IRP_MJ_FLUSH_BUFFERS
IRP_MJ_SHUTDOWN
IRP_MJ_DEVICE_CONTROL //with device-type-specific
                      //(public) I/O control codes
IRP_MJ_INTERNAL_DEVICE_CONTROL //with device-specific
                      //or driver-specific I/O control codes
```

As a general rule, an intermediate driver must handle at least the same IRP_MJ_*XXX* as the next-lower driver. That is, an intermediate driver must provide a Dispatch entry point for every IRP_MJ_*XXX* that the I/O Manager or a higher-level driver might send to the underlying device.

The specific operations a driver carries out for a given IRP_MJ_*XXX* depends somewhat on the underlying device, particularly for IRP_MJ_DEVICE_CONTROL and IRP_MJ_INTERNAL_DEVICE_CONTROL requests. However, the I/O Manager defines the parameters and I/O stack location contents for each major function code that it sets in IRPs.

The rest of this section summarizes request-specific information about parameters for the IRP_MJ_*XXX* that device and intermediate drivers handle, as well as general information about what kinds of drivers must handle each type of request and how they satisfy each type of request. See also *System Structures* for more information about the IRP, I/O stack location, and I/O status block structures.

The input and output parameters described in this section are the function-specific parameters in the IRP.

IRP_MJ_CLEANUP

Drivers that have Cancel routines also must handle cleanup requests.

Input Parameters

None

Output Parameters

None

When Sent

Receipt of this request indicates that the handle for the file object, representing the target device object, is being released.

Operation

If the driver's device objects were set up as exclusive, thereby allowing only a single thread to use the device at any given time, the driver completes every IRP currently queued to the target device object with STATUS_CANCELLED set in each IRP's I/O status block. Otherwise, the driver must cancel and complete any currently queued IRPs for the holder of the file object handle that is being released. After canceling outstanding IRPs in the queue, the driver completes the cleanup IRP with STATUS_SUCCESS set in its I/O status block.

IRP_MJ_CLOSE

Every driver must handle close requests, 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.

Input Parameters

None

Output Parameters

None

When Sent

Receipt of this request indicates that the handle of the file object that represents the target device object has been released.

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 create request. Device drivers whose device object(s) are exclusive, such as a serial driver, also can reset the hardware on receipt of a close request.

IRP_MJ_CREATE

Every kernel-mode driver must handle create requests.

Input Parameters

None

Output Parameters

None

When Sent

Receipt of this request indicates that a user-mode protected subsystem, possibly on behalf of an application, has requested a handle for the file object that represents the target device object, or that a higher-level driver is connecting or attaching its device object to the target device object.

Operation

Many device and intermediate drivers merely set STATUS_SUCCESS in the I/O status block of the IRP and complete the create request. However, what a given driver does on receipt of a create request depends on the driver's design. For example, a driver with pageable-image sections, like the system serial driver, maps in its paged-out code and allocates any resources necessary to handle subsequent I/O requests for the user-mode thread that is attempting to open the device for I/O.

IRP_MJ_DEVICE_CONTROL

Every driver whose device objects are of the same type is required to support this request. Every higher-level driver usually passes these requests on to an underlying device driver. Each device driver of a given type is assumed to support this request and a device-type-specific set of system-defined (also called *public*) I/O control codes (IOCTL_*XXX*) that determine what the driver does on receipt of this request.

Input Parameters

Depending on the I/O control code at **Parameters.DeviceIoControl.IoControlCode** in the driver's I/O stack location of the IRP, can be one of the following:

1. Caller-supplied data in the buffer at **Irp->AssociatedIrp.SystemBuffer**, together with the buffer's size in the I/O stack location at **Parameters.DeviceIoControl.Input-BufferLength** and/or a pointer to an additional input buffer in the I/O stack location at **Parameters.DeviceIoControl.Type3InputBuffer**.

2. Caller-supplied data in the buffer described by the MDL at **Irp->MdlAddress**, with the buffer's size in the I/O stack location at **Parameters.DeviceIoControl.InputBuffer-Length** if the lowest-level driver's device uses DMA or PIO and if the requested operation requires the transfer of a large amount of data quickly.
3. **Parameters.DeviceIoControl.OutputBufferLength** in the I/O stack location of the IRP indicates the size in bytes of the buffer into which the driver transfers data.

Output Parameters

Also determined by the I/O control code in the I/O stack location of the IRP

Data can be written into the buffer at **Irp->AssociatedIrp.SystemBuffer** or, using DMA or PIO, into the buffer described by the MDL at **Irp->MdlAddress**, as long as the transfer does not exceed the buffer's size.

When Sent

Any time following the successful completion of a create request

A 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.

Operation

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**.

IRP_MJ_FLUSH_BUFFERS

Drivers of devices with internal caches for data and drivers that maintain internal buffers for data must handle this request.

Input Parameters

None

Output Parameters

None

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.

Operation

The driver transfers any data currently cached in the device or held in the driver's internal buffer(s) 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.

IRP_MJ_INTERNAL_DEVICE_CONTROL

In general, any replacement for an existing driver that supports internal device control requests should handle this request. Such a driver also should 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.

Input Parameters

Depending on the I/O control code at **Parameters.DeviceIoControl.IoControlCode** in the I/O stack location of the IRP, can be one of the following:

1. Caller-supplied data in the buffer at **Irp->AssociatedIrp.SystemBuffer**, together with the buffer's size in the I/O stack location at **Parameters.DeviceIoControl.InputBufferLength** and/or a pointer to an additional input buffer in the I/O stack location at **Parameters.DeviceIoControl.Type3InputBuffer**.
2. Caller-supplied data in the buffer described by the MDL at **Irp->MdlAddress**, with the buffer's size in the I/O stack location at **Parameters.DeviceIoControl.InputBufferLength** if the lowest-level driver's device uses DMA or PIO and if the requested operation requires the transfer of a large amount of data quickly.

3. **Parameters.DeviceIoControl.OutputBufferLength** in the I/O stack location of the IRP indicates the size in bytes of the buffer into which the driver transfers data.

Output Parameters

Also determined by the I/O control code in the I/O stack location of the IRP

Data can be written into the buffer at **Irp->AssociatedIrp.SystemBuffer** or, using DMA or PIO, into the buffer described by the MDL at **Irp->MdlAddress**, as long as the transfer does not exceed the buffer's size.

When Sent

Any time after the successful completion of a create request.

For this request, the I/O control code (IOCTL_INTERNAL_*XXX*) 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.

Operation

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 IR_MJ_WRITE.

IRP_MJ_PNP

All drivers must be prepared to receive IRP_MJ_PNP requests.

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.

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.

Operation

See *Plug and Play IRPs* in Volume 1 of the *Windows 2000 Driver Development Reference* for detailed information about IRP_MJ_PNP requests.

IRP_MJ_POWER

All drivers must be prepared to receive IRP_MJ_POWER requests.

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.

When Sent

The Power Manager or a driver can send IRP_MJ_POWER requests at any time the operating system is running.

Operation

See *I/O Request for Power Management* Volume 1 of the *Windows 2000 Driver Development Reference* for detailed information about IRP_MJ_POWER requests.

IRP_MJ_READ

Every device driver that transfers data from its device to the system must handle read requests, as must any higher-level driver layered over such a device driver.

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.

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)

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.

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 IRP(s) 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.

IRP_MJ_SHUTDOWN

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.

Input Parameters

None

Output Parameters

None

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.

Operation

The driver must complete the transfer of any data currently cached in the device or held in the driver's internal buffer(s) before completing the shutdown request.

IRP_MJ_WRITE

Every device driver that transfers data from the system to its device must handle write requests, as must any higher-level driver layered over such a device driver.

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.

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

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.

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 IRP(s) 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.

Defining I/O Control Codes

All system-defined I/O control codes for IRP_MJ_DEVICE_CONTROL requests can be considered public in the sense that they are exported to one or more user-mode protected subsystems that run on top of the NT executive. As public device I/O control codes, some are also assumed to be exported to user-mode applications native to a protected subsystem, particularly to Win32 applications.

For a new kind of device, the driver's designer can define a public set of I/O control codes for IRP_MJ_DEVICE_CONTROL requests. However, since such a set of codes should be generally useful to other drivers of similar devices in the future, public I/O control codes must have the approval of and must be built into the system by Microsoft Corporation.

For new devices or for common kinds of devices with special features, driver designers also can define a set of I/O control codes for IRP_MJ_INTERNAL_DEVICE_CONTROL requests. Such a set of internal I/O control codes can be used by paired kernel-mode drivers to control the underlying device.

For example, kernel-mode drivers designed to the class/port model might use such a set of internal I/O control codes to take advantage of the special features of a particular device or type of device. The system-defined SCSI class/port interface uses this technique to define a SCSI-specific set of requests that class drivers send down to the system SCSI port driver, which transforms them into OS-independent SCSI requests for HBA-specific miniport drivers.

When it is sent a device control request, a class driver sets up the next-lower port driver's I/O stack location in the IRP and passes the request on to the underlying device driver, like any other higher-level driver.

A class driver also can allocate IRPs for I/O control requests and send them to the underlying port driver as follows:

1. Call **IoBuildDeviceIoControlRequest** to allocate an IRP with the major function code IRP_MJ_DEVICE_CONTROL or IRP_MJ_INTERNAL_DEVICE_CONTROL.

2. Set up the port driver's I/O stack location in the IRP with the IOCTL_*XXX* and appropriate parameters.
3. Note that parameters for a system-defined I/O control code almost never include an embedded pointer to avoid synchronization problems and possible access violations. With the exception of certain SCSI requests, the buffers at **Irp->AssociatedIrp.SystemBuffer**, at **Irp->MdlAddress**, and/or at **Parameters.DeviceIoControl.Type3InputBuffer** in the I/O stack location of the IRP neither have a pointer to another data buffer nor contain a structure with such a pointer for the public and system-defined internal I/O control codes.
4. 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 device driver. Such a pair of class/port drivers is responsible for ensuring that only one driver at a time can access the data and that their private data buffer is accessible in an arbitrary thread context by the port driver.
5. Call **IoSetCompletionRoutine** with the IRP, as necessary, so that the class driver can determine how the corresponding port driver handled a given request, reuse the IRP to send another request, or dispose of a driver-created IRP when the port driver completes a requested operation.
6. Call **IoCallDriver** to pass the request on to the port driver.

By calling the GDI function **EngDeviceIoControl**, a display driver also can 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 driver.

With a call to **DeviceIoControl**, a user-mode VDD can send I/O control requests to the corresponding kernel-mode driver for an MS-DOS®-application-dedicated device.

For more information about the functionality of video miniport drivers and display drivers, see the *Graphics Drivers Design Guide*. For additional VDD details see *Virtual Device Drivers* in the online DDK.

Figure 13.1 illustrates the layout of I/O control codes.

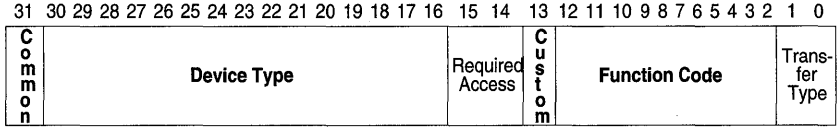


Figure 13.1 I/O Control Code Layout

Designers of drivers for a new FILE_DEVICE_*XXX* type of device must set the **Common** flag at bit 31 in the private I/O control codes they define. Those who define a private set of I/O control codes for IRP_MJ_DEVICE_CONTROL or IRP_MJ_INTERNAL_DEVICE_CONTROL requests also must set the **Custom** flag at bit 13 in the I/O control codes they define.

All system-defined I/O control codes have both these **C** flags cleared.

Driver writers can use the system-supplied macro CTL_CODE to set up new I/O control codes. To define an I/O control code, follow these guidelines for using CTL_CODE:

- Choose a descriptive constant name of the form IOCTL_Device_Function, where Device indicates the type of device and Function indicates the operation.
- Supply the following parameters to the CTL_CODE macro in the following order:
  1. *DeviceType* matches that set in the **Type** field of the driver's device objects.
  2. *FunctionCode* is in the range 0x800 to 0xfff for private I/O control codes defined by customers of Microsoft. Values in the range 0x000 to 0x7ff are reserved by Microsoft for public I/O control codes. The CTL_CODE macro automatically sets the **Custom** flag at bit 13 for values in the range 0x800 to 0xfff.
  3. *TransferType* indicates how data is passed to the driver as one of the following system-defined constants:
     - METHOD_BUFFERED if the driver transfers small amounts of data for the request. With this method, IRPs containing the I/O control code will supply a pointer to the buffer into which or from which to transfer data at **Irp->AssociatedIrp.SystemBuffer**. Most I/O control codes for device and intermediate drivers use this TransferType value.
     - METHOD_IN_DIRECT if the underlying device driver will read a large amount of data for the request using DMA or PIO and must transfer the data quickly. With this method, IRPs containing the I/O control code will supply a pointer to an MDL, describing the output buffer at **Irp->MdlAddress**.
     - METHOD_OUT_DIRECT if the underlying device driver will write a large amount of data to the device for the request using DMA or PIO and must transfer the data quickly. With this method, IRPs containing the I/O control code will supply a pointer to an MDL, describing the data buffer, at **Irp->MdlAddress**.

- METHOD_NEITHER if the driver can be sent such a request only while it is running in the context of the thread that originates the I/O control request. Only a highest-level kernel-mode driver is guaranteed to meet this condition, so this value is seldom used for the I/O control codes passed to 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. Otherwise, the originating user-mode caller might change the buffered data out from under the driver or the caller could be swapped out just as the driver is accessing the user buffer.

4. *RequiredAccess* indicates the type of access that must be requested when the caller opens the file object representing the device (see the IRP_MJ_CREATE request in *Input and Output Parameters for Common I/O Requests*). In other words, the I/O Manager will create IRPs and call the driver with a given I/O control code only if the caller has requested the necessary access rights for the driver to perform the requested operation. *RequiredAccess* can be one of the following system-defined constants:

   - FILE_ANY_ACCESS if the driver can carry out the requested operation for any caller that has a handle for the file object representing the target device object.
   - FILE_READ_DATA if the driver can carry out the requested operation only for a caller with read access rights. With this required access, the underlying device driver transfers data from the device to system memory.
   - FILE_WRITE_DATA if the driver can carry out the requested operation only for a caller with write access rights. With this required access, the underlying device driver transfers data from system memory to its device.
   - (FILE_READ_DATA | FILE_WRITE_DATA) if the caller must have both read and write access rights. With this required access, the underlying device driver transfers data between system memory and the device.

Most public I/O control requests sent to device drivers are assigned FILE_ANY_ACCESS as their *RequiredAccess* value, particularly those sent to drivers of exclusive devices and those that are buffered by the I/O Manager or a higher-level driver. Many internal I/O control requests for system-supplied drivers also specify this type of *RequiredAccess*.

However, for certain types of devices, the public I/O control codes require the caller to have read access rights, write access rights, or both.

For example, the definition of the public I/O control code IOCTL_DISK_SET_PARTITION_INFO shows that this I/O request can be sent to a disk driver and to all drivers layered above the disk driver only if the caller has both read and write access rights, as shown by the following definition:

```
#define IOCTL_DISK_SET_PARTITION_INFO\
   CTL_CODE(IOCTL_DISK_BASE, 0x008, METHOD_BUFFERED,\
         FILE_READ_DATA | FILE_WRITE_DATA)
```

Driver designers who want to define a set of public control codes must consult with Microsoft Corporation to have new codes added to the system header files. Private I/O control codes should be defined in the driver(s)' device-specific header files.

Device-type-specific I/O Requests

The remaining chapters in this document summarize the device-type-specific I/O requests handled by the system drivers of the most common kinds of devices.

Any 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 will replace a system driver for the same type of device.
- The new driver is for another device of a type already in the system.
- The new driver is an intermediate driver to be layered between two system drivers.

Such a new driver must handle every IRP_MJ_*XXX* that the system-supplied driver(s) handle. In most cases, a new device driver should also handle the same set of IOCTL_*XXX* 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.

The remainder of this document also supplies tips about the NTSTATUS values that drivers can set in the I/O status block of IRPs, set as necessary in an error log packet, and return for specific requests. These tips for selecting request-specific NTSTATUS values do not include STATUS_PENDING, which any driver can return for an IRP it has marked as pending (**IoMarkIrpPending**) and not yet completed. 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 the following kinds of drivers and the requests that each is required to support, see the following topics in the online DDK:

- *Serial Devices and Drivers*
- *Parallel Devices and Drivers*
- *Storage Drivers*
- *Microsoft Windows® 2000 Input Architecture*
- *Supporting USB Devices*
- *The IEEE 1394 Driver Stack*
- *Access Attribute Memory of a PCMCIA Device*
- *System Management Bus Driver Clients*

For all other types of drivers, consult the documentation for the appropriate driver type.

PART 2

Serial and Parallel Drivers

CHAPTER 1

Serial Driver Reference

This chapter describes the following topics about *Serial*, the Microsoft® Windows® 2000 system function driver for COM ports:

- *Serial Major I/O Requests*
- *Serial Device Control Requests*
- *Serial Internal Device Control Requests*

Serial is a function driver for legacy COM ports and Plug and Play COM ports. Serial is also a lower-level device filter driver for Plug and Play devices that require a 16550 UART-compatible interface, but are not attached to a COM port.

Serial implements the Serial service, and its executable image is *serial.sys*.

See the following for more information about Serial operation:

- *Serial Devices and Drivers* in the online DDK
- Data definitions in the include file *%install directory%\inc\ntddser.h* in the Windows 2000 DDK
- Sample code in the *%user install directory%\src\kernel\serial* directory in the Windows 2000 DDK
- Microsoft Win32® Communications API

Notes

1. This chapter describes Serial's operation as a function driver for COM ports. Serial's operation as a lower-level device filter driver is identical to its operation as a function driver.

2. Serial supports interfaces that are compatible with a 16550 UART. The device control signals and registers that are specified in this chapter are compatible with a 16550 UART interface. Examples of these device control signals and registers include: the *ready to send* control signal, the *clear to send* control signal, the *line control* register, and the *modem status* register.
3. This chapter does not duplicate the extensive information provided in the Microsoft Platform SDK about operating a COM port. For more information about the operation of Serial device control requests, see the corresponding documentation for the Win32 Communications API in the Platform SDK.

Serial Major I/O Requests

This section describes the Serial-specific operation of the following major I/O requests that Serial supports:

IRP_MJ_CLEANUP
IRP_MJ_CLOSE
IRP_MJ_CREATE
IRP_MJ_DEVICE_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_SYSTEM_CONTROL
IRP_MJ_WRITE

This section does not describe the driver-generic operation of these requests. See the following for more information on Serial's generic handling of these requests:

- *IRP Function Codes and IOCTLs*
- Sample code in the *%user install directory%\src\kernel\serial* directory in the Windows 2000 DDK

IRP_MJ_CREATE

Operation

The IRP_MJ_CREATE request opens a COM port. A COM port must be opened before it can be used. Only one client can have a COM port open at the same time.

I/O Status Block

The **Information** field is set to zero.

The **Status** field is set to one of the following values:

STATUS_SUCCESS

STATUS_ACCESS_DENIED

The device is already open.

STATUS_DELETE_PENDING

Serial is in the process of removing the device.

STATUS_INSUFFICIENT_RESOURCES

The device is not in a Plug and Play Started state, or the driver could not allocate an internal data structure.

STATUS_NOT_A_DIRECTORY

STATUS_PENDING

Serial queued the request for later processing.

STATUS_SHARED_IRQ_BUSY

The interrupt assigned to the device is in use by another open device.

IRP_MJ_DEVICE_CONTROL

Operation

Serial supports the following device control requests:

IOCTL_SERIAL_CLEAR_STATS
IOCTL_SERIAL_CLR_DTR
IOCTL_SERIAL_CLR_RTS
IOCTL_SERIAL_CONFIG_SIZE
IOCTL_SERIAL_GET_BAUD_RATE
IOCTL_SERIAL_GET_CHARS
IOCTL_SERIAL_GET_COMMSTATUS
IOCTL_SERIAL_GET_DTRRTS
IOCTL_SERIAL_GET_HANDFLOW
IOCTL_SERIAL_GET_LINE_CONTROL
IOCTL_SERIAL_GET_MODEM_CONTROL
IOCTL_SERIAL_GET_MODEMSTATUS
IOCTL_SERIAL_GET_PROPERTIES
IOCTL_SERIAL_GET_STATS

IOCTL_SERIAL_GET_TIMEOUTS
IOCTL_SERIAL_GET_WAIT_MASK
IOCTL_SERIAL_IMMEDIATE_CHAR
IOCTL_SERIAL_LSRMST_INSERT
IOCTL_SERIAL_PURGE
IOCTL_SERIAL_RESET_DEVICE
IOCTL_SERIAL_SET_BAUD_RATE
IOCTL_SERIAL_SET_BREAK_OFF
IOCTL_SERIAL_SET_BREAK_ON
IOCTL_SERIAL_SET_CHARS
IOCTL_SERIAL_SET_DTR
IOCTL_SERIAL_SET_FIFO_CONTROL
IOCTL_SERIAL_SET_HANDFLOW
IOCTL_SERIAL_SET_LINE_CONTROL
IOCTL_SERIAL_SET_MODEM_CONTROL
IOCTL_SERIAL_SET_QUEUE_SIZE
IOCTL_SERIAL_SET_RTS
IOCTL_SERIAL_SET_TIMEOUTS
IOCTL_SERIAL_SET_WAIT_MASK
IOCTL_SERIAL_SET_XOFF
IOCTL_SERIAL_SET_XON
IOCTL_SERIAL_WAIT_ON_MASK
IOCTL_SERIAL_XOFF_COUNTER

No other device control requests are supported.

For a description of the device control requests, see *Serial Device Control Requests*.

IRP_MJ_FLUSH_BUFFERS

Operation

The IRP_MJ_FLUSH_BUFFER request flushes the internal write buffer.

Serial queues and starts processing write and flush requests in the order in which the requests are received. Serial completes a flush request after it calls **IoCompleteRequest** for all write requests that it received before a flush request. However, completion of the flush request does not indicate that all the previously started write requests are completed by other drivers in the device stack. For example, a filter driver might still be processing a write request. A client must check that a write request is completed by all drivers in the device stack before the client attempts to free or reuse a write request's IRP.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following status values:

STATUS_SUCCESS

STATUS_CANCELLED

A client canceled the request. Serial also cancels a request if a device error occurs and Serial is configured to cancel a request if there is a device error.

STATUS_DELETE_PENDING

The driver is in the process of removing the device.

STATUS_PENDING

Serial queued the request for later processing.

IRP_MJ_INTERNAL_DEVICE_CONTROL

Serial supports the following internal device control requests:

IOCTL_SERIAL_INTERNAL_BASIC_SETTINGS
IOCTL_SERIAL_INTERNAL_CANCEL_WAIT_WAKE
IOCTL_SERIAL_INTERNAL_DO_WAIT_WAKE
IOCTL_SERIAL_INTERNAL_RESTORE_SETTINGS

No other internal device control requests are supported.

For a description of the internal device control requests, see *Serial Internal Device Control Requests*.

IRP_MJ_PNP

Serial supports the following Plug and Play requests:

IRP_MN_CANCEL_REMOVE_DEVICE
IRP_MN_CANCEL_STOP_DEVICE
IRP_MN_FILTER_RESOURCE_REQUIREMENTS
IRP_MN_QUERY_CAPABILITIES
IRP_MN_QUERY_DEVICE_RELATIONS
IRP_MN_QUERY_ID
IRP_MN_QUERY_PNP_DEVICE_STATE
IRP_MN_QUERY_REMOVE_DEVICE
IRP_MN_QUERY_RESOURCE_REQUIREMENTS x
IRP_MN_QUERY_STOP_DEVICE

IRP_MN_REMOVE_DEVICE
IRP_MN_START_DEVICE
IRP_MN_STOP_DEVICE
IRP_MN_SURPRISE_REMOVAL

Serial sends all other Plug and Play requests down the device stack without further processing.

Serial performs the following Serial-specific processing for Plug and Play requests:

IRP_MN_QUERY_ID (type BusQueryHardwardIDs)

If a COM port is on a multiport ISA card, Serial appends L"*PNP0502" to the string of hardware IDs.

IRP_MN_FILTER_RESOURCE_REQUIREMENTS

COM ports on a multiport ISA card share the same interrupt status register and the same interrupt.

For a description of the generic operation of Plug and Play requests, see *Plug and Play IRPs* in Volume 1 of the *Windows 2000 Driver Development Reference*.

IRP_MJ_POWER

Serial supports the following power requests:

IRP_MN_QUERY_POWER
IRP_MN_SET_POWER

Serial sends all other power requests down the device stack to be completed by a lower-level driver.

Serial is the default power policy owner for a serial device stack that uses Serial as a function driver or a lower-level filter driver.

For more information on the generic operation of these requests, see *I/O Request for Power Management* in Volume 1 of the *Windows 2000 Driver Development Reference*.

IRP_MJ_QUERY_INFORMATION

Operation

The IRP_MJ_QUERY_INFORMATION request queries the end-of-file information for a COM port. Serial supports requests of type **FileStandardInformation** and **FilePositionInformation**.

The standard file information is always set to zero or FALSE, as appropriate. The position information is always set to zero.

Input

The **Parameters.QueryFile.FileInformationClass** is set to **FileStandardInformation** or **FilePositionInformation**.

Output

FileStandardInformation

The **AssociatedIrp.SystemBuffer** member points to a client-allocated FILE_STANDARD_INFORMATION structure that Serial uses to output standard information.

FilePositionInformation

The **AssociatedIrp.SystemBuffer** member points to a client-allocated FILE_POSITION_INFORMATION structure that Serial uses to output position information.

Status I/O Block

If the request is successful, the **Information** member is set to zero.

The **Status** member is set to one of the following status values:

STATUS_SUCCESS

STATUS_CANCELLED

A client canceled the request. Serial also cancels a request if a device error occurs and Serial is configured to cancel a request if there is a device error.

STATUS_DELETE_PENDING

Serial is in the process of removing the device.

STATUS_INVALID_PARAMETER

The requested information is not supported.

STATUS_PENDING

Serial queued the request for later processing.

IRP_MJ_READ

Operation

The IRP_MJ_READ request copies bytes from a COM port to a client-allocated output buffer.

A client can use timeout events to terminate a read request. Note, however, that when a COM port is opened, the timeout settings for the device are undefined. A kernel-mode client can use an IOCTL_SERIAL_INTERNAL_BASIC_SETTINGS to set timeout parameters to

zero (no timeout events are used). User-mode and kernel-mode clients can use an IOCTL_SERIAL_SET_TIMEOUTS request to set timeout parameters.

For more information on read and write timeouts, see *Read and Write Timeouts for a COM Port* in Part 2, "Serial and Parallel Drivers," of the *Kernel-Mode Drivers Design Guide*.

Input

The **Parameters.Read.Length** member is set to the number of bytes to copy to the client's output buffer.

Output

The **AssociatedIrp.SystemBuffer** member points to a client-allocated buffer that Serial uses to output data.

Status I/O Block

The **Information** member is set to the number of bytes copied to the client's output buffer.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_CANCELLED

A client canceled the request. Serial also cancels a request if a device error occurs and Serial is configured to cancel a request if there is a device error.

STATUS_DELETE_PENDING

Serial is in the process of removing the device.

STATUS_PENDING

Serial queued the request for later processing.

STATUS_TIMEOUT

The time to complete the request exceeded the total timeout value or the interval timeout value.

IRP_MJ_SET_INFORMATION

Operation

The IRP_MJ_SET_INFORMATION request sets the end-of-file information on a COM port. Serial supports requests of type **FileEndOfFileInformation** and **FileAllocation-Information**.

A client can not set file information.

Input

The **Parameters.SetFile.FileInformationClass** member is set to **FileEndOfFile-Information** or **FileAllocationInformation**.

Status I/O Block

If the request is successful, the **Information** member is set to zero.

The **Status** member is set to one of the following status values:

STATUS_SUCCESS

STATUS_CANCELLED

A client canceled the request. Serial also cancels a request if a device error occurs and Serial is configured to cancel a request if there is a device error.

STATUS_DELETE_PENDING

Serial is in the process of removing the device.

STATUS_INVALID_PARAMETER

The specified end-of-file information is not supported.

STATUS_PENDING

Serial queued the request for later processing.

IRP_MJ_SYSTEM_CONTROL

Operation

Serial supports the following WMI requests:

IRP_MN_QUERY_ALL_DATA
IRP_MN_QUERY_DATA_BLOCK
IRP_MN_REGINFO

No other WMI requests or non-WMI system control requests are supported. If Serial does not support a WMI request, it skips the current stack location, and sends the request down the device stack.

Serial registers the following WMI GUIDS:

Serial WMI GUID	Associated Data Structure
SERIAL_PORT_WMI_NAME_GUID	USHORT followed by a WCSTR
SERIAL_PORT_WMI_COMM_GUID	SERIAL_WMI_COMM_DATA
SERIAL_PORT_WMI_HW_GUID	SERIAL_WMI_HW_DATA
SERIAL_PORT_WMI_PERF_GUID	SERIAL_WMI_PERF_DATA
SERIAL_PORT_WMI_PROPERTIES_GUID	WMI_SERIAL_PORT_PROPERTIES

The WMI name is the port name of the COM port, which is the value of the entry value **PortName** under the Plug and Play registry key for the device.

When called

WMI sends an IRP_MN_REGINFO request to Serial after Serial calls **IoWMIRegistrationControl** to update WMI registration information. Serial updates WMI registration information when a Plug and Play COM port is started and removed.

At the request of a WMI client, the WDM provider sends an IRP_MN_QUERY_DATA_BLOCK or an IRP_MN_QUERY_ALL_DATA request to obtain WMI data.

I/O Status Block

If the request is supported by the driver, the **Status** field is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The size in bytes of the output buffer is less than the size in bytes of the requested WMI data block.

STATUS_WMI_GUID_NOT_FOUND

The data block GUID is not valid.

STATUS_WMI_INSTANCE_NOT_FOUND

The WMI context is not valid.

If Serial does not handle the request, the **Status** member is set to STATUS_INVALID_DEVICE_REQUEST.

IRP_MJ_WRITE

Operation

The IRP_MJ_WRITE request copies data to a COM port from a client-provided input buffer.

A client can use timeout events to terminate a write request. Note, however, that when a COM port is opened, the timeout events set on a device are undefined. A kernel-mode client can use an IOCTL_SERIAL_INTERNAL_BASIC_SETTINGS to set timeout parameters to zero (no timeout events are used) and an IOCTL_SERIAL_SET_TIMEOUTS request to set timeout parameters. For more information on read and write timeouts, see *Read and Write Timeouts for a COM port* in Part 2, "Serial and Parallel Drivers," of the *Kernel-Mode Drivers Design Guide*.

Input

The **Parameters.Write.Length** member is set to the number of bytes to copy from a client-allocated input buffer to a COM port.

The **AssociatedIrp.SystemBuffer** member points to a client-allocated input buffer from which Serial copies data to the COM port.

Status I/O Block

The **Information** member is set to the number of bytes actually copied from the client's input buffer to the COM port.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_CANCELLED

A client canceled the request. Serial also cancels a request if a device error occurs and Serial is configured to cancel a request if there is a device error.

STATUS_DELETE_PENDING

Serial is in the process of removing the device.

STATUS_PENDING

Serial queued the request for later processing.

STATUS_TIMEOUT

The total time allowed for the write request was exceeded.

Serial Device Control Requests

This section describes the following device control requests that Serial supports:

IOCTL_SERIAL_CLEAR_STATS
IOCTL_SERIAL_CLR_DTR
IOCTL_SERIAL_CLR_RTS
IOCTL_SERIAL_CONFIG_SIZE
IOCTL_SERIAL_GET_BAUD_RATE
IOCTL_SERIAL_GET_CHARS
IOCTL_SERIAL_GET_COMMSTATUS
IOCTL_SERIAL_GET_DTRRTS
IOCTL_SERIAL_GET_HANDFLOW
IOCTL_SERIAL_GET_LINE_CONTROL
IOCTL_SERIAL_GET_MODEM_CONTROL
IOCTL_SERIAL_GET_MODEMSTATUS
IOCTL_SERIAL_GET_PROPERTIES
IOCTL_SERIAL_GET_STATS
IOCTL_SERIAL_GET_TIMEOUTS
IOCTL_SERIAL_GET_WAIT_MASK
IOCTL_SERIAL_IMMEDIATE_CHAR
IOCTL_SERIAL_LSRMST_INSERT
IOCTL_SERIAL_PURGE
IOCTL_SERIAL_RESET_DEVICE
IOCTL_SERIAL_SET_BAUD_RATE
IOCTL_SERIAL_SET_BREAK_OFF
IOCTL_SERIAL_SET_BREAK_ON
IOCTL_SERIAL_SET_CHARS
IOCTL_SERIAL_SET_DTR
IOCTL_SERIAL_SET_FIFO_CONTROL
IOCTL_SERIAL_SET_HANDFLOW
IOCTL_SERIAL_SET_LINE_CONTROL
IOCTL_SERIAL_SET_MODEM_CONTROL
IOCTL_SERIAL_SET_QUEUE_SIZE
IOCTL_SERIAL_SET_RTS
IOCTL_SERIAL_SET_TIMEOUTS
IOCTL_SERIAL_SET_WAIT_MASK
IOCTL_SERIAL_SET_XOFF
IOCTL_SERIAL_SET_XON
IOCTL_SERIAL_WAIT_ON_MASK
IOCTL_SERIAL_XOFF_COUNTER

No other device control codes are supported.

Status I/O Block

The **Information** member is set to a request-specific value or to zero.

The **Status** member is set to a request-specific status value or to one of the following generic status values:

STATUS_SUCCESS

STATUS_BUFFER_SIZE_TOO_SMALL

The input or the output buffer is too small to hold the required information.

STATUS_CANCELLED

The client canceled the request. For all control codes other than IOCTL_SERIAL_GET_COMMSTATUS, Serial also cancels a request if a device error occurs and Serial is configured to cancel a request if there is a device error.

STATUS_DELETE_PENDING

There was an unrecoverable hardware error, or a Plug and Play remove or surprise remove operation is in progress.

STATUS_INVALID_DEVICE_REQUEST

The COM port is not open.

STATUS_INVALID_PARAMETER

The request parameters are not valid.

STATUS_PENDING

Serial queued the request for later processing.

STATUS_Xxx

An internal operation returned an NTSTATUS error status value.

If Serial does not support a device control request, it sets the **Information** member to zero and the **Status** member to STATUS_INVALID_PARAMETER.

IOCTL_SERIAL_CLEAR_STATS

Operation

The IOCTL_SERIAL_CLEAR_STATS request clears the performance statistics for a COM port.

To obtain the performance statistics, a client can use an IOCTL_SERIAL_GET_STATS request.

Status I/O Block

The **Information** field is set to zero.

The **Status** field is set to one of the generic status values.

IOCTL_SERIAL_CLR_DTR

Operation

The IOCTL_SERIAL_CLR_DTR request clears the *data terminal ready* control signal (DTR).

If the handshake flow control of the device is configured to automatically use DTR, a client cannot clear or set DTR.

To set DTR, a client can use an IOCTL_SERIAL_SET_DTR request.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values. A status of STATUS_INVALID_PARAMETER indicates that the handshake flow control of the device is set to automatically use DTR.

IOCTL_SERIAL_CLR_RTS

Operation

The IOCTL_SERIAL_CLR_RTS request clears the *request to send* control signal (RTS).

If the handshake flow control of the device is configured to automatically use RTS, a client cannot clear or set RTS.

To set RTS, a client can use an IOCTL_SERIAL_SET_RTS request.

Status I/O Block

The **Information** member is set to zero.

The Status member is set to one of the generic status values. A status of STATUS_INVALID_PARAMETER indicates that the handshake flow control of the device is set to automatically use RTS.

IOCTL_SERIAL_CONFIG_SIZE

Operation

The IOCTL_SERIAL_CONFIG_SIZE request returns information about configuration size.

Serial always returns zero.

This request is obsolete and should not be used by new Microsoft® Windows® 2000 drivers.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a ULONG.

Output

The **AssociatedIrp.SystemBuffer** member points to a client-allocated ULONG buffer that Serial uses to output configuration size information.

Status I/O Block

The **Information** member is set to the size in bytes of a ULONG.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_BAUD_RATE

Operation

The IOCTL_SERIAL_GET_BAUD_RATE request returns the baud rate that is currently set for a COM port.

To set the baud rate, a client can use an IOCTL_SERIAL_SET_BAUD_RATE request.

For more information on the baud rates that the driver supports, see the baud rate constants SERIAL_BAUD_075 through SERIAL_BAUD_115200, which are defined in the include file *ntddser.h* in the Windows 2000 DDK.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a SERIAL_BAUD_RATE structure.

Output

The **AssociatedIrp.SystemBuffer** member points to a client-allocated SERIAL_BAUD_RATE structure that Serial uses to output the baud rate information.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a SERIAL_BAUD_RATE structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_CHARS

Operation

The IOCTL_SERIAL_GET_CHARS request returns the special characters that Serial uses with handshake flow control.

To set special characters, a client can use an IOCTL_SERIAL_SET_CHARS request.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a SERIAL_CHARS structure.

Output

The **AssociatedIrp.SystemBuffer** member points to a client-allocated SERIAL_CHARS structure that Serial uses to output the special characters.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of SERIAL_CHARS. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_COMMSTATUS

Operation

The IOCTL_SERIAL_GET_COMMSTATUS request returns information about the communication status of a COM port.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a SERIAL_STATUS structure.

Output

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_STATUS structure that Serial uses to output communication status information.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a SERIAL_STATUS structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_DTRRTS

Operation

The IOCTL_SERIAL_GET_DTRRTS request returns information about the *data terminal ready* control signal (DTR) and the *request to send* control signal (RTS).

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a ULONG.

Output

The **AssociatedIrp.SystemBuffer** points to a client-allocated ULONG buffer that Serial uses to output information about the DTR and RTS. The ULONG buffer is set to a logical OR of zero or more of the following flags:

SERIAL_DTR_STATE

Indicates that DTR is set.

SERIAL_RTS_STATE

Indicates that RTS is set.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a ULONG. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_HANDFLOW

Operation

The IOCTL_SERIAL_GET_HANDFLOW request returns information about the configuration of the handshake flow control set for a COM port.

To set the configuration of the handshake flow control, a client can use an IOCTL_SERIAL_SET_HANDFLOW request.

For more information on settings for handshake flow control, see the handshake flow control parameters SERIAL_DTR_MASK through SERIAL_FLOW_INVALID, which are defined in the include file *ntddser.h* in the Windows 2000 DDK.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a SERIAL_HANDFLOW structure.

Output

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_HANDFLOW structure that Serial uses to output information about the configuration of handshake flow control.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a SERIAL_HANDFLOW structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_LINE_CONTROL

Operation

The IOCTL_SERIAL_GET_LINE_CONTROL request returns information about the line control set for a COM port. The line control parameters include the number of stop bits, the number of data bits, and the parity.

To configure the line control, a client can use an IOCTL_SERIAL_SET_LINE_CONTROL request.

For information on valid line control register settings, see the constants SERIAL_5_DATA through SERIAL_PARITY_MASK, which are defined in the include file *%install directory%\src\kernel\serial.h* in the Windows 2000 DDK.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a SERIAL_LINE_CONTROL structure.

Output

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_LINE_CONTROL structure that Serial uses to output information about the line control configuration.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a SERIAL_LINE_CONTROL structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_MODEM_CONTROL

Operation

The IOCTL_SERIAL_GET_MODEM_CONTROL request returns the value of the modem control register.

To set the modem control register, a client can use an IOCTL_SERIAL_SET_MODEM_ CONTROL request.

For information on modem control register settings, see the constants SERIAL_MCR_DTR through SERIAL_MCR_LOOP, which are defined in the include file *%install directory%\src\kernel)\serial\serial.h* in the Windows 2000 DDK.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a ULONG.

Output

The **AssociatedIrp.SystemBuffer** points to a client-allocated ULONG buffer that Serial uses to output the value of the modem control register.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a ULONG. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_MODEMSTATUS

Operation

The IOCTL_SERIAL_GET_MODEMSTATUS request updates the modem status, and returns the value of the modem status register before the update.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a ULONG.

Output

The **AssociatedIrp.SystemBuffer** points to a client-allocated ULONG buffer that Serial uses to output the value of the modem status register.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a ULONG. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_PROPERTIES

Operation

The IOCTL_SERIAL_GET_PROPERTIES request returns information about the capabilities of a COM port.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a SERIAL_COMMPROP structure.

Output

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_COMMPROP structure that Serial uses to output information about the capabilities of the COM port.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a SERIAL_COMMPROP structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_STATS

Operation

The IOCTL_SERIAL_GET_STATS request returns information about the performance of a COM port. The statistics include the number of characters transmitted, the number of characters received, and useful error statistics. The driver continuously increments performance values.

To reset the performance values to zero, a client can use an IOCTL_SERIAL_CLEAR_STATS request.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a SERIALPERF_STATS structure.

Output

The **AssociatedIrp.SystemBuffer** member points to a client-allocated SERIALPERF_STATS structure that Serial uses to output performance information.

Status I/O Block

If the request succeeds, the **Information** member is set to the size in bytes of a SERIALPERF_STATS structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_TIMEOUTS

Operation

The IOCTL_SERIAL_GET_TIMEOUTS request returns the timeout values that Serial uses with read and write requests.

To set timeouts, a client can use an IOCTL_SERIAL_SET_TIMEOUTS request.

For more information on timeouts, see *Read and Write Timeouts for a COM Port* in Part 2, "Serial and Parallel Drivers," of the *Kernel-Mode Drivers Design Guide*.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a SERIAL_TIMEOUTS structure.

Output

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_TIMEOUTS structure that Serial uses to output information about read and write timeout values.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a SERIAL_TIMEOUTS structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_GET_WAIT_MASK

Operation

The IOCTL_SERIAL_GET_WAIT_MASK request returns the event wait mask that is currently set on a COM port.

A client can wait on the wait events SERIAL_EV_RXCHAR through SERIAL_EV_EVENT2 that are defined in the include file *ntddser.h* in the Windows 2000 DDK.

To set an event wait mask, a client can use an IOCTL_SERIAL_SET_WAIT_MASK request. To wait for the occurrence of a wait event, a client uses an IOCTL_SERIAL_WAIT_ON_MASK request.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a ULONG.

Output

The **AssociatedIrp.SystemBuffer** points to a client-allocated ULONG buffer that Serial uses to output the wait mask. The wait mask is zero or a logical OR of one or more of the wait events SERIAL_EV_RXCHAR through SERIAL_EV_EVENT2 that are defined in the include file *ntddser.h* in the Windows 2000 DDK.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a ULONG. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_IMMEDIATE_CHAR

Operation

The IOCTL_SERIAL_IMMEDIATE_CHAR request causes a specified character to be transmitted as soon as possible. The immediate character request completes immediately after any other write that might be in progress. Only one immediate character request can be pending at a time.

Input

The **AssociatedIrp.SystemBuffer** points to the UCHAR value to transmit immediately.

The **Parameters.DeviceIoControl.InputBufferLength** is set to the size in bytes of an UCHAR.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a UCHAR. Otherwise, **Information** is set to zero.

The **Status** member is set to one of generic status values. A status of STATUS_INVALID_PARAMETER indicates that a previous immediate character request is pending.

IOCTL_SERIAL_LSRMST_INSERT

Operation

The IOCTL_SERIAL_LSRMST_INSERT request enables or disables the insertion of information about line status and modem status in the receive data stream. If LSRMST insertion is enabled, the driver inserts event information for the supported event types. The event information includes an *event header* followed by event-specific data. The event header contains a client-specified escape character and a flag that identifies the event. The driver supports the following event types:

SERIAL_LSRMST_LSR_DATA

A change occurred in the line status. Serial inserts an event header followed by the event-specific data, which is the value of the line status register followed by the character present in the receive hardware when the line-status change was processed.

SERIAL_LSRMST_LSR_NODATA

A line status change occurred, but no data was available in the receive buffer. Serial inserts an event header followed by the event-specific data, which is the value of the line status register when the line status change was processed.

SERIAL_LSRMST_MST

A change occurred in the modem status. Serial inserts an event header followed by the event-specific data, which is the value of the modem status register when the modem-status change was processed.

SERIAL_LSRMST_ESCAPE

Indicates that the next character in the receive data stream, which was received from the device, is identical to the client-specified escape character. Serial inserts an event header. There is no event-specific data.

Input

The **AssociatedIrp.SystemBuffer** member points to a client-allocated UCHAR that is used to input an escape character. If the escape character is nonNULL, insertion is enabled, and the serial driver uses the specified escape character. Otherwise, insertion is disabled.

The **Parameters.DeviceIoControl.InputBufferLength** is set to the size in bytes of a UCHAR.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a UCHAR. Otherwise, **Information** is set to zero.

The **Status** member is set to one of the generic status values. A status of STATUS_INVALID_PARAMETER indicates that the specified escape character is the same as the XON or the XOFF character, or that error replacement is enabled with handshake flow control.

IOCTL_SERIAL_PURGE

Operation

The IOCTL_SERIAL_PURGE request cancels the specified requests and deletes data from the specified buffers. The purge request can be used to cancel all read requests and write requests and to delete all data from the read buffer and the write buffer.

The completion of the purge request does not indicate that the requests cancelled by the purge request are completed. A client must verify that the purged requests are completed before the client frees or reuses the corresponding IRPs.

Input

The **AssociatedIrp.SystemBuffer** member points to a client-allocated ULONG that is used to input a *purge mask*. The client sets the purge mask to a logical OR of one or more of the following purge flags:

SERIAL_PURGE_RXABORT

Purges all read requests.

SERIAL_PURGE_RXCLEAR

Purges the receive buffer, if one exists.

SERIAL_PURGE_TXABORT

Purges all write requests.

SERIAL_PURGE_TXCLEAR

Purges the write buffer, if one exists.

The **Parameters.DeviceIoControl.InputBufferLength** is set to the size in bytes of a ULONG.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a ULONG. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the generic status values. A status of STATUS_INVALID_PARAMETER indicates that the purge mask is not valid.

IOCTL_SERIAL_RESET_DEVICE

Operation

The IOCTL_SERIAL_RESET_DEVICE request resets a COM port.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_SET_BAUD_RATE

Operation

The IOCTL_SERIAL_SET_BAUD_RATE request sets the baud rate on a COM port. Serial verifies the specified baud rate.

To obtain the baud rate, a client can use an IOCTL_SERIAL_GET_BAUD_RATE request.

For more information on the baud rates that the driver supports, see the baud rate constants SERIAL_BAUD_075 through SERIAL_BAUD_115200, which are defined in the include file *ntddser.h* in the Windows 2000 DDK.

Input

The **AssociatedIrp.SystemBuffer** member points to a SERIAL_BAUD_RATE structure that a client allocates and sets to input the baud rate.

The **Parameters.DeviceIoControl.InputBufferLength** member is set to the size in bytes of a SERIAL_BAUD_RATE structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_SET_BREAK_OFF

Operation

The IOCTL_SERIAL_SET_BREAK_OFF request sets the line control break signal inactive.

To set the line control break signal active, a client can use an IOCTL_SERIAL_SET_BREAK_ON request.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is to one of the generic status values.

IOCTL_SERIAL_SET_BREAK_ON

Operation

The IOCTL_SERIAL_SET_BREAK_ON request sets the line control break signal active.

To set the line control break signal inactive, a client can use an IOCTL_SERIAL_SET_BREAK_OFF request.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_SET_CHARS

Operation

The IOCTL_SERIAL_GET_CHARS request sets the special characters that Serial uses for handshake flow control. Serial verifies the specified special characters.

To obtain the special characters, a client can use an IOCTL_SERIAL_GET_CHARS request.

Input

The **AssociatedIrp.SystemBuffer** member points to a client-allocated SERIAL_CHARS structure that is used to input special characters.

The **Parameters.DeviceIoControl.InputBufferLength** member is set to the size in bytes of a SERIAL_CHARS structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic values. A status of STATUS_INVALID_PARAMETER indicates that **XoffChar** equals **XonChar**, or that one of them equals the handshake flow control escape character.

IOCTL_SERIAL_SET_DTR

Operation

The IOCTL_SERIAL_SET_DTR request sets DTR.

If the handshake flow control of the device is configured to automatically use DTR, a client can not clear or set DTR.

To clear DTR, a client can use an IOCTL_SERIAL_CLR_DTR request.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic values. A status of STATUS_INVALID_PARAMETER indicates that the handshake flow control of the device is set to automatically use DTR.

IOCTL_SERIAL_SET_FIFO_CONTROL

Operation

The IOCTL_SERIAL_SET_INFO_CONTROL request sets the FIFO control register (FCR). Serial does not verify the specified FIFO control information.

Input

The **AssociatedIrp.SystemBuffer** member points to a client-allocated ULONG that is used to input FIFO control information.

The **Parameters.DeviceIoControl.InputBufferLength** is set to the size in bytes of a ULONG.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_SET_HANDFLOW

Operation

The IOCTL_SERIAL_SET_HANDFLOW request sets the configuration of handshake flow control. Serial verifies the specified handshake flow control information.

To obtain handshake flow control information, a client can use an IOCTL_SERIAL_GET_ HANDFLOW request..

For more information on settings for handshake flow control, see the handshake flow control parameters SERIAL_DTR_MASK through SERIAL_FLOW_INVALID, which are defined in the include file *ntddser.h* in the Windows 2000 DDK.

Input

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_HANDFLOW structure that is used to input the handshake flow control information.

The **Parameters.DeviceIoControl.InputBufferLength** is set to the size in bytes of a SERIAL_HANDFLOW structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic values. A status of STATUS_INVALID_ PARAMETER indicates the specified handshake flow control is not valid.

IOCTL_SERIAL_SET_LINE_CONTROL

Operation

The IOCTL_SERIAL_SET_LINE_CONTROL request sets the *line control register* (LCR). The line control register controls the data size, the number of stop bits, and the parity.

To obtain the value of the line control register, a client can use an IOCTL_SERIAL_GET_LINE_CONTROL request.

For information on valid line control register settings, see the constants SERIAL_5_DATA through SERIAL_PARITY_MASK, which are defined in the include file *%install directory%\src\kernel\serial.h* in the Windows 2000 DDK.

Input

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_LINE_CONTROL structure that is used to input line control information.

The **Parameters.DeviceIoControl.InputBufferLength** is set to the size in bytes of a SERIAL_LINE_CONTROL structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values. A status of STATUS_INVALID_PARAMETER indicates that the specified line control information is not valid.

IOCTL_SERIAL_SET_MODEM_CONTROL

Operation

The IOCTL_SERIAL_SET_MODEM_CONTROL request sets the modem control register. Parameter checking is not done.

To obtain the value of the modem control register, a client can use an IOCTL_SERIAL_GET_MODEM_CONTROL request.

For information on modem control register settings, see the constants SERIAL_MCR_DTR through SERIAL_MCR_LOOP, which are defined in the include file *%install directory%\src\kernel\serial\serial.h* in the Windows 2000 DDK.

Input

The **AssociatedIrp.SystemBuffer** points to a client-allocated ULONG that is used to input modem control information.

The **Parameters.DeviceIoControl.InputBufferLength** is set to the size in bytes of a ULONG.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_SET_QUEUE_SIZE

Operation

The IOCTL_SERIAL_SET_QUEUE_SIZE request sets the size of the internal receive buffer. If the requested size is greater than the current receive buffer size, a new receive buffer is created. Otherwise, the receive buffer is not changed.

Input

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_QUEUE_SIZE structure that is used to input a receive buffer size.

The **Parameters.DeviceIoControl.InputBufferLength** is set to the size in bytes of a SERIAL_QUEUE_SIZE structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_SET_RTS

Operation

The IOCTL_SERIAL_SET_RTS request sets RTS.

If a handshake flow control of the device is configured to automatically use RTS, a client can not clear or set RTS.

A client can use an IOCTL_SERIAL_CLR_RTS request to clear RTS.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values. A status of STATUS_INVALID_PARAMETER indicates that the handshake flow control of the device is set to automatically use RTS.

IOCTL_SERIAL_SET_TIMEOUTS

Operation

The IOCTL_SERIAL_SET_TIMEOUTS request sets the timeout values that the driver uses with read and write requests.

To obtain the timeout values, a client can use an IOCTL_SERIAL_GET_TIMEOUTS request.

For more information on timeouts, see *Read and Write Timeouts for a COM Port* in Part 2, "Serial and Parallel Drivers," of the *Kernel-Mode Drivers Design Guide*.

Input

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_TIMEOUTS structure that is used to input read and write timeout values.

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a SERIAL_TIMEOUTS structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values. A status of STATUS_INVALID_PARAMETER indicates that the read timeout values exceed the maximum permitted values.

IOCTL_SERIAL_SET_WAIT_MASK

Operation

The IOCTL_SERIAL_SET_WAIT_MASK request configures Serial to notify a client after the occurrence of any one of a specified set of *wait events*.

A client can wait on the events SERIAL_EV_RXCHAR through SERIAL_EV_*EVENT2*, which are defined in the include file *ntddser.h* in the Windows 2000 DDK. A client specifies wait events by setting an input *event wait mask* to a logical OR of one or more of the event flags. A client can clear all wait events by setting the input event wait mask to zero.

A client uses an IOCTL_SERIAL_WAIT_ON_MASK request to wait for the occurrence of a wait event. If a wait-on-mask request is already pending when a set-wait-mask request is processed, the pending wait-on-event request is completed with a status of STATUS_SUCCESS and the output wait event mask is set to zero.

Input

The **AssociatedIrp.SystemBuffer** points to a ULONG buffer that the client allocates and sets to an event wait mask. The wait mask is set to zero or a logical OR of one or more of the event flags.

The **Parameters.DeviceIoControl.InputBufferLength** is set to the size in bytes of a ULONG.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values. A status of STATUS_INVALID_PARAMETER indicates that the input wait mask is not valid.

IOCTL_SERIAL_SET_XOFF

Operation

The IOCTL_SERIAL_SET_XOFF request emulates the reception of an XOFF character. The request stops reception of data. If automatic XON/XOFF flow control is not set, then a client *must* use a subsequent IOCTL_SERIAL_SET_XON request to restart reception of data.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_SET_XON

Operation

The IOCTL_SERIAL_SET_XON request emulates the reception of a XON character, which restarts reception of data.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values.

IOCTL_SERIAL_WAIT_ON_MASK

Operation

The IOCTL_SERIAL_WAIT_ON_MASK request is used to wait for the occurrence of any wait event specified by using an IOCTL_SERIAL_SET_WAIT_MASK request. A wait-on-mask request is completed after one of the following events occurs:

- A wait event occurs that was specified by the most recent set-wait-mask request.
- An IOCTL_SERIAL_SET_WAIT_MASK request is received while a wait-on-mask request is pending. The driver completes the pending wait-on-mask request with a status of STATUS_SUCCESS and the output wait mask is set to zero.

A client can wait on the events SERIAL_EV_RXCHAR through SERIAL_EV_EVENT2, which are defined in the include file *ntddser.h* in the Windows 2000 DDK.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a ULONG.

Output

AssociatedIrp.System buffer points to a ULONG buffer that holds an event wait mask. The event wait mask indicates which wait events occurred. The event wait mask is set to zero or a logical OR of one or more of the wait mask flags.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values. A status of STATUS_INVALID_PARAMETER indicates that no wait events are set, or a wait-on-mask request is already pending.

IOCTL_SERIAL_XOFF_COUNTER

Operation

The IOCTL_SERIAL_XOFF_COUNTER request sets an *XOFF counter*. An XOFF counter request supports clients that use software to emulate hardware handshake flow control.

An XOFF counter request is synchronized with write requests. The driver sends a specified XOFF character, and completes the request after one of the following events occurs:

- A write request is received.
- A timer expires (a timeout value is specified by the XOFF counter request).
- Serial receives a number of characters that is greater than or equal to a count specified by the XOFF counter request.

For more information about the operation of an XOFF counter, see the description of the SERIAL_XOFF_COUNTER structure in the include file *ntddser.h* in the Windows 2000 DDK.

Input

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_XOFF_COUNTER structure that is used to input XOFF counter information.

The **Parameters.DeviceIoControl.InputBufferLength** is set to the size in bytes of a SERIAL_XOFF_COUNTER structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the generic status values or to one of the following request-specific values:

STATUS_INVALID_PARAMETER

The count value specified for the XOFF counter request is less than zero.

STATUS_SERIAL_MORE_WRITES

A write request was received.

STATUS_SERIAL_COUNTER_TIMEOUT

Serial Internal Device Control Requests

This section describes the following internal device control requests that Serial supports:

IOCTL_SERIAL_INTERNAL_BASIC_SETTINGS
IOCTL_SERIAL_INTERNAL_CANCEL_WAIT_WAKE
IOCTL_SERIAL_INTERNAL_DO_WAIT_WAKE
IOCTL_SERIAL_INTERNAL_RESTORE_SETTINGS

No other internal device control requests are supported.

These requests are provided for trusted kernel-mode drivers.

Status I/O Block

The setting of the status I/O block members is request-specific. If Serial does not support the request, it sets the **Information** member to zero and the **Status** member to STATUS_INVALID_PARAMETER.

IOCTL_SERIAL_INTERNAL_BASIC_SETTINGS

Operation

The IOCTL_SERIAL_INTERNAL_BASIC_SETTINGS request sets a COM port to a basic operating mode. Serial's basic operating mode reads and writes one byte at a time, and does not use handshake flow control or timeouts. The basic operation mode is suitable for use by a driver that uses a subset of the 16550 UART interface. Examples of such drivers include a mouse driver or a graphics pad driver for older hardware that use a 16450 UART.

The IOCTL_SERIAL_INTERNAL_BASIC_SETTINGS request returns the operating mode settings that are in use just before Serial sets the basic operation mode. A client uses an IOCTL_SERIAL_INTERNAL_RESTORE_SETTINGS request to restore a previous operating mode. A client should treat the operating mode settings as opaque. Serial does not verify the settings when the settings are restored. Note also that a replacement driver for Serial might implement a different set of basic settings.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a SERIAL_BASIC_SETTINGS structure.

Output

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_BASIC_SETTINGS structure that Serial uses to output the current configuration.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of SERIAL_BASIC_SETTINGS. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of **Parameters.DeviceIoControl.OutputLength** is less than the size in bytes of a SERIAL_BASIC_SETTINGS structure.

STATUS_CANCELLED

STATUS_DELETE_PENDING

STATUS_PENDING

IOCTL_SERIAL_INTERNAL_CANCEL_WAIT_WAKE

Operation

The IOCTL_SERIAL_INTERNAL_CANCEL_WAIT_WAKE request disables the wait/wake operation of a COM port.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following status values:

STATUS_SUCCESS

STATUS_CANCELLED

STATUS_DELETE_PENDING

STATUS_PENDING

IOCTL_SERIAL_INTERNAL_DO_WAIT_WAKE

Operation

The IOCTL_SERIAL_INTERNAL_DO_WAIT_WAKE request enables the wait/wake operation of a COM port.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following status values:

STATUS_SUCCESS

STATUS_CANCELLED

STATUS_DELETE_PENDING

STATUS_PENDING

IOCTL_SERIAL_INTERNAL_RESTORE_SETTINGS

Operation

The IOCTL_SERIAL_INTERNAL_RESTORE_SETTINGS request restores the specified operating mode of a COM port. The specified operating mode should be a mode that was returned by an IOCTL_SERIAL_INTERNAL_BASIC_SETTINGS request. The operating mode settings should be treated as opaque. Serial does not verify the settings when the settings are restored. Note also that a replacement for Serial might implement a different set of parameters.

Input

The **AssociatedIrp.SystemBuffer** points to a client-allocated SERIAL_BASIC_SETTINGS structure that is used to input operating mode settings. The client should use settings that were returned by an IOCTL_SERIAL_INTERNAL_BASIC_SETTINGS request.

The **Parameters.DeviceIoControl.OutputBufferLength** is set to the size in bytes of a SERIAL_BASIC_SETTINGS structure.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of SERIAL_BASIC_SETTINGS structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The **Parameters.DeviceIoControl.InputLength** is less than the size in bytes of a SERIAL_BASIC_SETTINGS structure.

STATUS_CANCELLED

STATUS_DELETE_PENDING

STATUS_PENDING

CHAPTER 2

Serenum Driver Reference

This chapter describes the following topics about *Serenum*, a Microsoft® Windows® 2000 system filter driver for RS-232 ports:

- *Serenum Device Control Requests*
- *Serenum Internal Device Control Requests*

Serenum is a Plug and Play upper-level device filter driver that enumerates the following device types:

- Plug and Play serial devices that comply with *Plug and Play External COM Device Specification, Version 1.00, February 28, 1995*
- Pointer devices that comply with legacy mouse detection in Windows NT® 4 and earlier

Serenum is used with *Serial*, the Windows 2000 system function driver for COM ports. The combined operation of Serial and Serenum acts as a Plug and Play bus driver for a RS-232 port. Serenum can also be used with other RS-232 port drivers.

Serenum implements the Serenum service, and its executable image is *serenum.sys*.

See the following topics for more information about Serenum operation:

- *Serial Devices and Drivers* in the online DDK
- Data definitions in the include file *%install directory%\inc\ntddkser.h* in the Windows 2000 DDK
- Sample code in the *%user install directory%\src\kernel\serenum* directory in the Windows 2000 DDK
- Include file *%install directory%\src\kernel\serial\serial.h* in the Windows 2000 DDK

Serenum Device Control Requests

This section describes the following device control requests that Serenum supports for a filter device object, or *filter DO*:

IOCTL_SERENUM_PORT_DESC
IOCTL_SERENUM_GET_PORT_NAME

For all other device control requests, Serenum skips the current IRP stack location, and sends the request down the device stack without further processing.

If Serenum receives a device control request for a physical device object, it routes the request to the filter DO.

IOCTL_SERENUM_PORT_DESC

Operation

The IOCTL_SERENUM_PORT_DESC request returns a description of the RS-232 port associated with a filter DO.

Input

Parameters.DeviceIoControl.InputBufferLength and **Parameters.DeviceIoControl. OutputBufferLength** are set to the size in bytes of a SERENUM_PORT_DESC structure.

The **AssociatedIrp.SystemBuffer** member is set to a pointer to a client-allocated buffer that is used to input and output a SERENUM_PORT_DESC structure. The client *must* set the **Size** member of the input structure to the size in bytes of a SERENUM_PORT_DESC structure.

Output

The **AssociatedIrp.SystemBuffer** member points to the client-allocated buffer that Serenum uses to output a SERENUM_PORT_DESC structure. Serenum sets the following members:

PortHandle

Specifies a pointer to the filter DO.

PortAddress

Specifies the base physical address of the RS-232 port.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a SERENUM_PORT_DESC structure.

The **Status** member is set to one of the following values:

STATUS_SUCCESSSTATUS_DELETE_PENDING

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength, Parameters.DeviceIoControl. OutputBufferLength, or the **Size** member of the input structure is not equal to the size in bytes of a SERENUM_PORT_DESC structure.

IOCTL_SERENUM_GET_PORT_NAME

Operation

The IOCTL_SERENUM_GET_PORT_NAME request returns the value of the **PortName** (or **Identifer**) entry value for the RS-232 port.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a client-allocated output buffer.

Output

The **AssociatedIrp.SystemBuffer** member points to a client-allocated buffer that Serenum uses to output the port name. The port name is a zero-terminated Unicode string.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of the zero-terminated Unicode string that is returned in the client's output buffer.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The output buffer is too small to hold the port name string.

STATUS_UNSUCCESSFUL

An error occurred when opening the registry key for the device or reading the **PortName** (or **Identifier**) entry value.

Serenum Internal Device Control Requests

This section describes the following internal device control requests that Serenum supports for a physical device object, or *PDO*:

- IOCTL_INTERNAL_SERENUM_REMOVE_SELF

Serenum sends all other internal device control requests down the device stack of the filter DO associated with the PDO. Serenum performs no further processing of the request. The request is completed by a lower-level driver in the device stack of the filter DO.

If Serenum receives an internal device control request for a filter DO, Serenum sends the request down the device stack of the filter DO without further processing.

IOCTL_INTERNAL_SERENUM_REMOVE_SELF

Operation

The IOCTL_INTERNAL_SERENUM_REMOVE_SELF request invalidates the bus relations of the filter DO associated with a target PDO. (Physically, this request invalidates the bus relations of the RS-232 port that the target device is attached to.)

Status I/O Block

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_DELETE_PENDING

CHAPTER 3

Parport Driver Reference

This chapter includes the following topics about *Parport*, the Microsoft® Windows® 2000 system driver for parallel ports:

- Parport Major I/O Requests
- Parport Internal Device Request Control Requests
- Parport Data types
- Parport Callback Routines

Parport is a function driver for *parallel port controllers*, commonly referred to as *parallel ports*. The interface class of a parallel port is GUID_PARALLEL_CLASS. The Parport service is in the Parallel arbitrator group of services. The executable image of Parport is *parport.sys*.

For more information on Parport operation, see:

- *Parallel Devices and Drivers* in the online DDK
- Sample code in the *%install directory%\src\kernel\parport* directory in the Windows 2000 DDK
- Include files *%install directory%\inc\ddk\parallel.h* and *%install directory%\inc\ntddpar.h* in the Windows 2000 DDK

Parport Major I/O Requests

This section describes the Parport-specific handling of the following major I/O requests that Paport supports:

IRP_MJ_CLEANUP
IRP_MJ_CLOSE

IRP_MJ_CREATE
IRP_MJ_INTERNAL_DEVICE_CONTROL
IRP_MJ_PNP
IRP_MJ_POWER
IRP_MJ_SYSTEM_CONTROL

See the following topics for more information on Parport's generic handling of these major I/O requests:

- *IRP Function Codes and IOCTLs*
- Sample Parclass code in the *%install directory%\src\kernel\parclass* directory of the Windows 2000 DDK
- *Plug and Play IRPs* in Volume 1
- *I/O Requests for Power Management* in Volume 1
- *WMI IRPs* in the online DDK

IRP_MJ_CREATE

Operation

The IRP_MJ_CREATE request opens a parallel port. Parport increments the count of open files on a parallel port. A parallel port is a shared device.

Note that Parclass opens each parallel port after the port's interface is enabled. Parport registers and enables an GUID_PARALLEL_CLASS interface for each parallel port that is enumerated in the system.

I/O Status Block

The **Information** member is set to zero.

The **Status** member is set to one of the following:

STATUS_SUCCESS

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

IRP_MJ_INTERNAL_DEVICE_CONTROL

Operation

Parport supports the following internal device control requests:

IOCTL_INTERNAL_DESELECT_DEVICE
IOCTL_INTERNAL_GET_MORE_PARALLEL_PORT_INFO
IOCTL_INTERNAL_GET_PARALLEL_PNP_INFO
IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO
IOCTL_INTERNAL_INIT_1284_3_BUS
IOCTL_INTERNAL_PARALLEL_CLEAR_CHIP_MODE
IOCTL_INTERNAL_PARALLEL_CONNECT_INTERRUPT
IOCTL_INTERNAL_PARALLEL_DISCONNECT_INTERRUPT
IOCTL_INTERNAL_PARALLEL_PORT_ALLOCATE
IOCTL_INTERNAL_PARALLEL_PORT_FREE
IOCTL_INTERNAL_PARALLEL_SET_CHIP_MODE
IOCTL_INTERNAL_RELEASE_PARALLEL_PORT_INFO
IOCTL_INTERNAL_SELECT_DEVICE

No other internal device control requests are supported.

For more information on these internal device control requests, see *Parport Internal Device Control Requests* in this chapter.

I/O Status Block

The values of the status block members are request-specific. If the request is not supported, the **Information** member is set to zero and the **Status** member is set to STATUS_INVALID_PARAMETER.

Parport Internal Device Control Requests

This section describes the following topics:

IOCTL_INTERNAL_DESELECT_DEVICE
IOCTL_INTERNAL_GET_MORE_PARALLEL_PORT_INFO
IOCTL_INTERNAL_GET_PARALLEL_PNP_INFO
IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO
IOCTL_INTERNAL_INIT_1284_3_BUS
IOCTL_INTERNAL_PARALLEL_CLEAR_CHIP_MODE
IOCTL_INTERNAL_PARALLEL_CONNECT_INTERRUPT
IOCTL_INTERNAL_PARALLEL_DISCONNECT_INTERRUPT

IOCTL_INTERNAL_PARALLEL_PORT_ALLOCATE
IOCTL_INTERNAL_PARALLEL_PORT_FREE
IOCTL_INTERNAL_PARALLEL_SET_CHIP_MODE
IOCTL_INTERNAL_RELEASE_PARALLEL_PORT_INFO
IOCTL_INTERNAL_SELECT_DEVICE

No other internal device control requests are supported. If the request is not supported, the **Information** member is set to zero and the **Status** member is set to STATUS_INVALID_PARAMETER.

IOCTL_INTERNAL_DESELECT_DEVICE

Operation

The IOCTL_INTERNAL_DESELECT_DEVICE request deselects an IEEE 1284.3 daisy-chain device or an IEEE 1284 end-of-chain device on a parallel port. The client should have the port allocated. In addition to deselecting a device, the client can also free the port.

Input

The **AssociatedIrp.SystemBuffer** member points to a PARALLEL_1284_COMMAND structure that the client allocates to input IEEE 1284.3 command information. The client can free the port by not setting the PAR_HAVE_PORT_KEEP_PORT flag in the **Command-Flags** member.

The **Parameters.DeviceIoControl.InputBufferLength** member specifies the size in bytes of the PARALLEL_1284_COMMAND structure.

Status I/O Block

The **Information** member is set to zero.

The Status member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.InputBufferLength** member is less than the size in bytes of a PARALLEL_1284_COMMAND structure.

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

STATUS_INVALID_PARAMETER

The specified device is not flagged internally as an end-of-chain device, and the specified ID value is greater than the number of existing daisy-chain devices.

STATUS_UNSUCCESSFUL

The device state is invalid and unknown.

IOCTL_INTERNAL_GET_MORE_PARALLEL_PORT_INFO

Operation

The IOCTL_INTERNAL_GET_MORE_PARALLEL_PORT_INFO request returns information about a parallel port. This information supplements the information that a client obtains by using an IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO request. The port information includes the type of system interface, the bus number, and the interrupt resources used by the parallel port.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a MORE_PARALLEL_PORT_INFORMATION structure

Output

The **AssociatedIrp.SystemBuffer** member points to a MORE_PARALLEL_PORT_INFORMATION structure that the client allocates to output parallel port information.

Status I/O Block

If the request succeeds, the **Information** member is set to the size in bytes of the MORE_PARALLEL_PORT_INFORMATION structure. Otherwise; the **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.OutputBufferLength** member is less than the size in bytes of a MORE_PARALLEL_PORT_INFORMATION structure.

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

IOCTL_INTERNAL_GET_PARALLEL_PNP_INFO

Operation

The IOCTL_INTERNAL_GET_PARALLEL_PNP_INFO request returns Plug and Play information about a parallel port.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a PARALLEL_PNP_INFORMATION structure.

Output

The **AssociatedIrp.SystemBuffer** member points to a PARALLEL_PNP_INFORMATION structure that the client allocates to output Plug and Play information.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of **Parameters.DeviceIoControl.OutputBufferLength** is less than the size in bytes of a PARALLEL_PNP_INFORMATION structure.

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO

Operation

The IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO request returns information about a parallel port. The information specifies the resources assigned to the parallel port and the capabilities of the parallel port. The structure also contains pointers to callback routines that a kernel-mode driver can use to operate the parallel port.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a PARALLEL_PORT_INFORMATION structure.

Output

The **AssociatedIrp.SystemBuffer** member points to a PARALLEL_PORT_INFORMATION structure that the client allocates to output the parallel port information.

Status I/O Block

If this request succeeds, the **Information** member is set to the size in bytes of a PARALLEL_PORT_INFORMATION structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.OutputBufferLength** member is less than the size in bytes of a PARALLEL_PORT_INFORMATION structure.

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

IOCTL_INTERNAL_INIT_1284_3_BUS

Operation

The IOCTL_INTERNAL_INIT_1284_3_BUS request initializes and assigns an IEEE 1284.3 device ID to all the devices that are attached to the port. A port can have up to four daisy-chain devices and one end-of-chain device attached. Daisy-chain device IDs are integers with a range of zero to three. An end-of-chain device has an ID of four.

Note that this request is included primarily for test purposes.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

IOCTL_INTERNAL_PARALLEL_CLEAR_CHIP_MODE

Operation

The IOCTL_INTERNAL_PARALLEL_CLEAR_CHIP_MODE request clears the operating mode of a parallel port. To clear the mode, the operating mode specified by this

request must match the current operating mode. The clear request works in conjunction with the set mode request. To set a new operating mode, the current operating mode must first be cleared.

Note that a kernel-mode driver can also clear the operating mode of the parallel port by using the **ClearChipMode** callback routine and can set the operating mode by using the **TrySetChipMode** callback routine.

Input

The **AssociatedIrp.SystemBuffer** member points to a PARALLEL_CHIP_MODE structure that the client allocates to input chip mode information. The client sets the **ModeFlags** member to the current operating mode.

The request sets the **Parameters.DeviceIoControl.InputBufferLength** member to the size in bytes of a PARALLEL_CHIP_MODE structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.InputBufferLength** member is less than the size in bytes of a PARALLEL_CHIP_MODE structure.

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

STATUS_INVALID_DEVICE_STATE

The specified operating mode is not the same as the current operating mode of the parallel port.

IOCTL_INTERNAL_PARALLEL_CONNECT_INTERRUPT

Operation

The IOCTL_INTERNAL_PARALLEL_CONNECT_INTERRUPT request connects an optional interrupt service routine and an optional *deferred port check* routine to a parallel port. This request can only be used by kernel-mode drivers.

Note that Microsoft does not recommend using a client-supplied interrupt routine. Using a client-supplied interrupt routine might cause system instability. By default, the connection of an interrupt service routine is disabled. The design for using interrupts in a Plug and Play environment and in an IEEE 1284.3 daisy-chain environment is still under development.

The connect interrupt request returns information that the driver can use in the context of a driver-specific ISR. The information includes a pointer to the interrupt object and pointers to callback routines that allocate and free the parallel port at IRQL DIRQL.

Parport maintains a list of the ISRs that are connected to a parallel port. Parport calls all the connected ISRs after an interrupt on the parallel port.

Parport also maintains a list of optional deferred port check routines that are connected to a parallel port. Parport calls all deferred port check routines after the parallel port is freed and there are no requests queued on the parallel port work queue. Parport allocates the parallel port before calling the deferred port check routines, and then frees the parallel port after all the port check routines return.

The connect interrupt request is enabled by the registry entry value **EnableConnect-InterruptIoctl** under the Plug and Play registry key for the parallel port device. The type of the entry value is REG_DWORD and the default value is 0x0 (disabled).

Input

The **AssociatedIrp.SystemBuffer** member points to a PARALLEL_INTERRUPT_SERVICE_ROUTINE structure that the client allocates to input interrupt service information. Note that Parport uses the same memory buffer, but casts it to a different data type to output information.

The **Parameters.DeviceIoControl.InputBufferLength** member is set to the size in bytes of a PARALLEL_INTERRUPT_SERVICE_ROUTINE structure.

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a PARALLEL_INTERRUPT_INFORMATION structure.

Output

The **AssociatedIrp.SystemBuffer** member points to a PARALLEL_INTERRUPT_INFORMATION structure that Parport uses to output parallel interrupt information.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a PARALLEL_INTERRUPT_INFORMATION structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

At least one of the following is true:

The value of the **Parameters.DeviceIoControl.InputBufferLength** member is less than the size in bytes of a PARALLEL_INTERRUPT_SERVICE_ROUTINE structure.

The value of the **Parameters.DeviceIoControl.OuputBufferLength** member is less than the size in bytes of a PARALLEL_INTERRUPT_INFORMATION structure.

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

STATUS_INSUFFICIENT_RESOURCES

Parport could not allocate a dynamic data structure.

STATUS_UNSUCCESSFUL

The connect interrupt request is disabled.

IOCTL_INTERNAL_PARALLEL_DISCONNECT_INTERRUPT

Operation

The IOCTL_INTERNAL_PARALLEL_DISCONNECT_INTERRUPT request disconnects an interrupt service routine (and an optional deferred port check service routine) that was connected by using an IOCTL_INTERNAL_PARALLEL_CONNECT_INTERRUPT request. This request is only used by kernel-mode drivers.

Note that Microsoft does not recommend using a client-supplied interrupt routine. Using a client-supplied interrupt routine might cause system instability. By default, the connection of an interrupt service routine is disabled. The design for using interrupts in a Plug and Play environment and in an IEEE 1284.3 daisy-chain environment is still under development.

Input

The **AssociatedIrp.SystemBuffer** member points to a PARALLEL_INTERRUPT_SERVICE_ROUTINE structure that the client allocates for the input of interrupt service information.

The **Parameters.DeviceIoControl.InputBufferLength** member is set to the size in bytes of a PARALLEL_INTERRUPT_SERVICE_ROUTINE structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.InputBufferLength** member is less than the size in bytes of a PARALLEL_INTERRUPT_SERVICE_ROUTINE structure.

STATUS_INVALID_PARAMETER

The specified interrupt service routine is not connected.

IOCTL_INTERNAL_PARALLEL_PORT_ALLOCATE

Operation

The IOCTL_INTERNAL_PARALLEL_PORT_ALLOCATE request allocates a parallel port. Before using a parallel port, a client must first allocate the port. If the port is already allocated, Parport marks the allocate request as pending, and adds the allocate request to the parallel port work queue.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_CANCELLED

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

STATUS_PENDING

The request was added to the work queue of the parallel port.

IOCTL_INTERNAL_PARALLEL_PORT_FREE

Operation

The IOCTL_INTERNAL_PARALLEL_PORT_FREE request frees a parallel port. A client must free a parallel port after it is finished using the parallel port. If there are no requests

pending on the port's work queue, Parport calls the deferred port check routines that were connected by IOCTL_INTERNAL_PARALLEL_CONNECT_INTERRUPT requests.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

IOCTL_INTERNAL_PARALLEL_SET_CHIP_MODE

Operation

The IOCTL_INTERNAL_PARALLEL_SET_CHIP_MODE request sets the operating mode of a parallel port.

To set the operating mode, the operating mode must first be cleared. A client can use an IOCTL_INTERNAL_PARALLEL_CLEAR_CHIP_MODE request to clear the mode. Note that a kernel-mode driver can also use the **ClearChipMode** callback routine to clear the mode.

Input

The **AssociatedIrp.SystemBuffer** member points to a PARALLEL_CHIP_MODE structure that the client allocates to input chip mode information. The client sets the **ChipMode** member to the requested operating mode. For more information on the operating modes, see the ECR modes that are defined in *%Windows 2000 install directory%\inc\ddk\parallel.h*

The **Parameters.DeviceIoControl.InputBufferLength** member is set to the size, in bytes, of a PARALLEL_CHIP_MODE structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.InputBufferLength** member is less than the size, in bytes, of a PARALLEL_CHIP_MODE structure.

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

STATUS_INVALID_DEVICE_STATE

The mode is not cleared.

STATUS_NO_SUCH_DEVICE

The requested operating mode is not valid.

IOCTL_INTERNAL_RELEASE_PARALLEL_PORT_INFO

Operation

The IOCTL_INTERNAL_RELEASE_PARALLEL_PORT_INFO request returns STATUS_SUCCESS without further processing.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

IOCTL_INTERNAL_SELECT_DEVICE

Operation

The IOCTL_INTERNAL_SELECT_DEVICE request allocates a port and selects an IEEE 1284.3 daisy-chain device or an IEEE 1284 end-of-chain device on a parallel port. If the client already allocated the parallel port, then it can request Parport to only select the device. If the parallel port is not allocated and cannot immediately be allocated, the select device request is added to the work queue of the parallel port.

Input

The **AssociatedIrp.SystemBuffer** points to a PARALLEL_1284_COMMAND structure that the client allocates to input select device information.

The **Parameters.DeviceIoControl.InputBufferLength** member specifies the size in bytes of a PARALLEL_1284_COMMAND structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.InputBufferLength** member is less than the size in bytes of a PARALLEL_1284_COMMAND structure.

STATUS_CANCELLED

The client does not have the port allocated and the select request is canceled before the port driver attempts to allocate the port.

STATUS_DELETE_PENDING

The device is in a Plug and Play surprised-removed state.

STATUS_INVALID_PARAMETER

The specified device is not flagged internally as an end-of-chain device and the value of the **ID** member of the input structure is greater than the number of existing daisy-chain devices.

STATUS_PENDING

The client does not have the port allocated and there are other requests pending on the port work queue. The select request is added to the port work queue.

IOCTL_INTERNAL_Xxx

Operation

Internal device control requests that are not documented in this chapter are completed without further processing.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to STATUS_INVALID_PARAMETER.

Parport Data Types

This section describes the following data types that Parport uses to input and output information:

MORE_PARALLEL_PORT_INFORMATION

Specifies information about the system interface and the interrupt that support the operation of a parallel port.

PARALLEL_1284_COMMAND

Specifies information that a client uses to select and deselect an IEEE 1284.3 device or an IEEE 1284 end-of-chain device.

PARALLEL_CHIP_MODE

Specifies the operating mode of a parallel port.

PARALLEL_PNP_INFORMATION

Specifies information about the capabilities of a parallel port.

PARALLEL_PORT_INFORMATION

Specifies information about the resources assigned to a parallel port and the capabilities of a parallel port. The structure also contains pointers to callback routines that a kernel-mode driver can use to operate the parallel port.

PARALLEL_INTERRUPT_INFORMATION

Specifies information that a kernel-mode driver can use in the context of an ISR that the driver connects to a parallel port.

PARALLEL_INTERRUPT_SERVICE_ROUTINE

Specifies interrupt services that a kernel-mode driver can connect to the operation of a parallel port.

MORE_PARALLEL_PORT_INFORMATION

```
typedef struct _MORE_PARALLEL_PORT_INFORMATION {
    INTERFACE_TYPE  InterfaceType;
    ULONG           BusNumber;
    ULONG           InterruptLevel;
    ULONG           InterruptVector;
    KAFFINITY       InterruptAffinity;
    KINTERRUPT_MODE InterruptMode;
} MORE_PARALLEL_PORT_INFORMATION, *PMORE_PARALLEL_PORT_INFORMATION;
```

The MORE_PARALLEL_PORT_INFORMATION structure specifies information about the system interface and the parallel port interrupt.

Members

InterfaceType

Specifies the interface type associated with the parallel port.

BusNumber

Specifies the bus number for the interface.

InterruptLevel

Specifies the interrupt level for the parallel port.

InterruptVector

Specifies the interrupt vector for the parallel port.

InterruptAffinity

Specifies the interrupt affinity value.

InterruptMode

Specifies the interrupt mode.

Include

parallel.h

Comments

An IRP_MN_START_DEVICE request from the Plug and Play Manager passes a translated resource list that contains the information in a MORE_PARALLEL_PORT_INFORMATION structure. Parport saves the information, in the extension of the device object that represents the parallel port, and returns the information in response to an IOCTL_INTERNAL_GET_MORE_PARALLEL_PORT_INFO request.

See Also

PARALLEL_PORT_INFORMATION, IOCTL_INTERNAL_GET_PARALLEL_PNP_INFO, IOCTL_INTERNAL_GET_MORE_PARALLEL_PORT_INFO, IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO

PARALLEL_1284_COMMAND

```
typedef struct _PARALLEL_1284_COMMAND {
    UCHAR    ID;
    UCHAR    Port;
    ULONG    CommandFlags;
} PARALLEL_1284_COMMAND, *PPARALLEL_1284_COMMAND;
```

The PARALLEL_1284_COMMAND structure specifies information that a client uses to select and deselect an IEEE 1284.3 daisy-chain device or an IEEE 1284 end-of-chain device.

Members

ID

Specifies the IEEE 1284.3 device ID. Parport assigns integer IDs to daisy-chain devices in a range of zero to three.

Port

Reserved (set to zero).

CommandFlags

Specifies a bitwise OR of zero or more of the following flags:

PAR_END_OF_CHAIN_DEVICE

Specifies an end-of-chain device.

PAR_HAVE_PORT_KEEP_PORT

Specifies that the client has the port allocated, and makes a request to keep the port allocated.

Include

parallel.h

Comments

A parallel port supports the connection of zero to four IEEE 1284.3 daisy-chain devices and a single IEEE 1284 end-of-chain device. The end-of-chain device must be an IEEE 1284 device but does not have to be an IEEE 1284.3 device. Parport assigns integer IDs to daisy-chain devices in a range of zero to three.

See Also

IOCTL_INTERNAL_DESELECT_DEVICE, IOCTL_INTERNAL_SELECT_DEVICE, **DeselectDevice**, **TrySelectDevice**

PARALLEL_CHIP_MODE

```
typedef struct _PARALLEL_CHIP_MODE {
    UCHAR       ModeFlags;
    BOOLEAN     success;
} PARALLEL_CHIP_MODE, *PPARALLEL_CHIP_MODE;
```

The PARALLEL_CHIP_MODE structure specifies the operating mode of a parallel port.

Members

ModeFlags

Specifies an operating mode of a parallel port (an EPP or ECP mode).

Success

Not used.

Include

parallel.h

Comments

A client uses a PARALLEL_CHIP_MODE structure with internal device control requests to set and clear the operating mode of a parallel port.

See Also

IOCTL_INTERNAL_PARALLEL_CLEAR_CHIP_MODE, IOCTL_INTERNAL_PARALLEL_SET_CHIP_MODE

PARALLEL_PNP_INFORMATION

```
typedef struct _PARALLEL_PNP_INFORMATION {
    PHYSICAL_ADDRESS                OriginalEcpController;
    PUCHAR                          EcpController;
    ULONG                           SpanOfEcpController;
    ULONG                           PortNumber;
    ULONG                           HardwareCapabilities;
    PPARALLEL_SET_CHIP_MODE         TrySetChipMode;
    PPARALLEL_CLEAR_CHIP_MODE       ClearChipMode;
    ULONG                           FifoDepth;
```

```
    ULONG                           FifoWidth;
    PHYSICAL_ADDRESS                EppControllerPhysicalAddress;
    ULONG                           SpanOfEppController;
    ULONG                           Ieee1284_3DeviceCount;
    PPARALLEL_TRY_SELECT_ROUTINE    TrySelectDevice;
    PPARALLEL_DESELECT_ROUTINE      DeselectDevice;
    PVOID                           Context;
    ULONG                           CurrentMode;
    PWSTR                           PortName;         // symbolic link name
                                                      // for legacy device object
} PARALLEL_PNP_INFORMATION, *PPARALLEL_PNP_INFORMATION;
```

The PARALLEL_PNP_INFORMATION structure specifies information about the capabilities of a parallel port.

Members

OriginalEcpController

Specifies the base physical address that Parport uses to control the ECP operation of the parallel port.

EcpController

Pointer to the I/O port resource that is used to control the port in ECP mode.

SpanOfEcpController

Specifies the size in bytes of the I/O port resource.

PortNumber

Not used.

HardwareCapabilities

Specifies the hardware capabilities of the parallel port. The following capabilities can be set using a bitwise OR of the following constants:

PPT_1284_3_PRESENT
PPT_BYTE_PRESENT
PPT_ECP_PRESENT
PPT_EPP_32_PRESENT
PPT_EPP_PRESENT
PT_NO_HARDWARE_PRESENT

TrySetChipMode

Pointer to a callback routine that a kernel-mode driver can use to change the operating mode of the parallel port.

ClearChipMode

Pointer to a callback routine that a kernel-mode driver can use to clear the operating mode of the parallel port,

FifoDepth

Specifies the size in bytes of the hardware FIFO.

FifoWidth

Specifies the width, in bits, of the FIFO (number of bits handled in parallel).

EppControllerPhysicalAddress

Not used.

SpanOfEppController

Not used.

Ieee1284_3DeviceCount

Specifies the number of daisy-chain devices currently on the parallel port. A range of zero to four such devices can be simultaneously connected to a parallel port.

TrySelectDevice

Pointer to a callback routine that a kernel-mode driver can use to try to select an IEEE 1284.3 device.

DeselectDevice

Pointer to a callback routine that a kernel-mode driver can use to deselect an IEEE 1284.3 device.

Context

Pointer to the extension of the device object that represents the parallel port.

CurrentMode

The current operating mode of the parallel port.

PortName

The symbolic link name of the parallel port.

Include

parallel.h

See Also

IOCTL_INTERNAL_GET_PARALLEL_PNP_INFO

PARALLEL_PORT_INFORMATION

```
typedef struct _PARALLEL_PORT_INFORMATION {
    PHYSICAL_ADDRESS                OriginalController;
    PUCHAR                          Controller;
    ULONG                           SpanOfController;
    PPARALLEL_TRY_ALLOCATE_ROUTINE  TryAllocatePort
    PPARALLEL_FREE_ROUTINE          FreePort
    PPARALLEL_QUERY_WAITERS_ROUTINE QueryNumWaiters
    PVOID                           Context
} PARALLEL_PORT_INFORMATION, *PPARALLEL_PORT_INFORMATION;
```

The PARALLEL_PORT_INFORMATION structure specifies information about the resources that are assigned to a parallel port. The structure also contains pointers to callback routines that a kernel-mode driver can use to operate the parallel port.

Members

OriginalController

Specifies the bus relative base I/O address of the parallel port registers.

Controller

Pointer to the system-mapped base I/O location of the parallel port registers.

SpanOfController

Specifies the size in bytes of the I/O space allocated to the parallel port.

TryAllocatePort

Pointer to a callback routine that a kernel-mode driver can use to try to allocate the parallel port.

FreePort

Pointer to a callback routine that that a kernel-mode driver can use to free the parallel port.

QueryNumWaiters

Pointer to a callback routine that a kernel-mode driver can use to determine the number of requests on the work queue of the parallel port.

Context

Pointer to the device extension of the device object that represents a parallel port.

Include

parallel.h

Comments

An IRP_MN_START_DEVICE request from the PnP Manager passes a translated resource list that contains the port information in a PARALLEL_PORT_INFORMATION structure. Parport saves the information in the extension of the device object that represents the parallel port and returns the information in response to an IOCTL_GET_PARALLEL_PORT_INFO request.

See Also

IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO

PARALLEL_INTERRUPT_INFORMATION

```
typedef struct _PARALLEL_INTERRUPT_INFORMATION {
    PKINTERRUPT                     InterruptObject;
    PPARALLEL_TRY_ALLOCATE_ROUTINE  TryAllocatePortAtInterruptLevel;
    PPARALLEL_FREE_ROUTINE          FreePortFromInterruptLevel;
    PVOID                           Context;
} PARALLEL_INTERRUPT_INFORMATION, *PPARALLEL_INTERRUPT_INFORMATION;
```

The PARALLEL_INTERRUPT_INFORMATION structure specifies information that a kernel-mode driver can use in the context of an ISR that the driver connects to a parallel port.

Members

InterruptObject

Pointer to the parallel port interrupt object.

TryAllocatePortAtInterruptLevel

Pointer to a callback routine that a kernel-mode driver can use to try to allocate the parallel port at IRQL DIRQL.

FreePortFromInterruptLevel

Pointer to a callback routine that a kernel-mode driver can use to free the parallel port at IRQL DIRQL.

Context

Pointer to the extension of the device object that represents the parallel port.

Include

parallel.h

Comments

A kernel-mode driver can use the parallel interrupt information in the context of an ISR. A driver connects an interrupt service routine using an IOCTL_INTERNAL_PARALLEL_CONNECT_INTERRUPT request.

See Also

IOCTL_INTERNAL_PARALLEL_CONNECT_INTERRUPT

PARALLEL_INTERRUPT_SERVICE_ROUTINE

```
typedef struct _PARALLEL_INTERRUPT_SERVICE_ROUTINE {
 .PKSERVICE_ROUTINE           InterruptServiceRoutine;
 .PVOID                       InterruptServiceContext;
 .PPARALLEL_DEFERRED_ROUTINE  DeferredPortCheckRoutine;
 .PVOID                       DeferredPortCheckContext;
} PARALLEL_INTERRUPT_SERVICE_ROUTINE, *PPARALLEL_INTERRUPT_SERVICE_ROUTINE;
```

A PARALLEL_INTERRUPT_SERVICE_ROUTINE structure specifies interrupt services that a kernel-mode driver can connect to the operation of a parallel port.

Members

InterruptServiceRoutine

Pointer to an interrupt service routine.

InterruptServiceContext

Pointer to a context for the interrupt service routine.

DeferredPortCheckRoutine

Pointer to an optional deferred port check routine:

```
VOID
(*DeferredPortCheckRoutine) (
    IN PVOID DeferredContext
);
```

Parameters

DeferredContext

Pointer to a context for the deferred port check routine.

DeferredPortCheckContext

Pointer to an optional context for the deferred port check routine.

Include

parallel.h

Comments

A kernel-mode driver can connect a device-specific interrupt service routine and a deferred port check routine to the parallel port.

See Also

IOCTL_INTERNAL_PARALLEL_CONNECT_INTERRUPT

Parport Callback Routines

This section describes the Parport callback routines that a kernel-mode driver can use to operate a parallel port. Note that most of the Parport callback routines have equivalent device control requests, which are preferred over using the following callback routines:

ClearChipMode

Clears the operating mode of a parallel port.

DeselectDevice

Deselects an IEEE 1284.3 daisy-chain device or an IEEE 1284 end-of-chain device. A caller can also specify that Parport free the parallel port.

FreePort

Frees a parallel port.

FreePortFromInterruptLevel

Frees a parallel port at IRQL DIRQL.

QueryNumWaiters

Returns the number of requests queued on the work queue of a parallel port.

TryAllocatePort

Allocates a parallel port.

TryAllocatePortAtInterruptLevel

Allocates a parallel port at IRQL DIRQL.

TrySelectDevice

Selects an IEEE 1284.3 daisy-chain device or an IEEE 1284 end-of-chain device on a parallel port.

TrySetChipMode

Sets the operating mode of a parallel port.

ClearChipMode

```
NTSTATUS
(*ClearChipMode) (

IN PDEVICE_EXTENSION Extension,
  IN UCHAR ChipMode
)
```

The **ClearChipMode** callback routine clears the operating mode of a parallel port.

Parameters

Extension

Pointer to the extension of the device object that represents the parallel port.

ChipMode

Specifies the current operating mode of the parallel port.

For more information on operating modes, see the ECR modes defined in *%Windows 2000 DDK install directory%\inc\ddk\parallel.h*.

Include

parallel.h

Return Value

STATUS_SUCCESS

STATUS_INVALID_DEVICE_STATE

The specified mode does not match the current mode.

Comments

A kernel-mode driver uses an IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO request to obtain the **ClearChipMode** pointer.

The **ClearChipMode** callback routine clears the operating mode of the parallel port. A caller uses **ClearChipMode** in conjunction with **TrySetChipMode**.

To set a mode, a caller must first clear the current mode.

To clear the current mode, a caller must specify the same mode that was used to set the current mode.

See Also

IOCTL_INTERNAL_PARALLEL_CLEAR_CHIP_MODE, IOCTL_INTERNAL_PARALLEL_SET_CHIP_MODE, **TrySetChipMode**

DeselectDevice

```
NTSTATUS
PptDeselectDevice(
  IN PVOID Context,
  IN PVOID DeselectCommand
)
```

The **DeselectDevice** callback routine deselects an IEEE 1284.3 daisy-chain device or an IEEE 1284 end-of-chain device. A caller can also specify that Parport free the parallel port.

Parameters

Context

Pointer to the extension of the device object that represents the parallel port.

DeselectCommand

Pointer to a PARALLEL_1284_COMMAND structure. The caller specifies the following parameters:

ID

Specifies the 1284.3 device ID.

CommandFlags

Specifies a bitwise OR of zero or more of the following flags:

Value	Description
PAR_END_OF_CHAIN_DEVICE	Specifies an end-of-chain device.
PAR_HAVE_PORT_KEEP_PORT	Specifies that the port be kept allocated.

Include

parallel.h

Return Value

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

The specified device ID is invalid.

STATUS_UNSUCCESSFUL

Parport could not deselect the device.

Comments

A kernel-mode driver uses an IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO request to obtain the **DeselectDevice** pointer.

A kernel-mode driver can use an IOCTL_INTERNAL_DESELECT_DEVICE request or a **DeselectDevice** call to deselect a device on a parallel port. To deselect a device, a caller should have the port allocated. If the caller does not set the PAR_HAVE_PORT_KEEP_PORT flag, Parport frees the port after deselecting the device.

DeSelectDevice runs in the caller's thread at IRQL <= DISPATCH_LEVEL.

See Also

IOCTL_INTERNAL_SELECT_DEVICE, **TrySelectDevice**

FreePort

```
VOID
(*FreePort)(
  IN PVOID Context
)
```

The **FreePort** callback routine frees a parallel port.

Parameters

Context

Pointer to the device extension of the device object that represents the parallel port.

Include

parallel.h

Comments

A kernel-mode driver uses an IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO request to obtain the **FreePort** pointer.

The driver should allocate a parallel port before freeing it. A driver can use **QueryNumWaiters** to determine the number of clients that are waiting to allocate the parallel port, and **TryAllocatePort** to try to allocate the parallel port.

FreePort runs in the caller's thread at the IRQL of the caller.

See Also

FreePortFromInterruptLevel, **QueryNumWaiters**, **TryAllocatePort**, **TryAllocatePortAtInterruptLevel**

FreePortFromInterruptLevel

```
VOID
(*FreePortFromInterruptLevel)(
  IN PVOID Context
)
```

The **FreePortFromInterruptLevel** callback routine frees a parallel port at IRQL DIRQL.

Parameters

Context

Pointer to the extension of the device object that represents the parallel port.

Include

parallel.h

Comments

A kernel-mode driver connects an interrupt service routine by using an IOCTL_INTERNAL_PARALLEL_CONNECT_INTERRUPT to obtain a **FreePortFromInterruptLevel** pointer.

The driver should allocate the port before freeing it. A driver can use **TryAllocatePortAtInterruptLevel** to try to allocate the port at IRQL DIRQL.

If there are no requests on the work queue, **FreePortFromInterruptLevel** immediately frees the port; otherwise, it queues a deferred procedure call that frees the port at a later time.

FreePortFromInterruptLevel runs at IRQL DIRQL.

See Also

TryAllocatePortAtInterruptLevel

QueryNumWaiters

```
ULONG
(*QueryNumWaiters)(

IN PVOID Extension
)
```

The **QueryNumWaiters** callback routine returns the number of requests that are queued on the work queue of a parallel port.

Parameters

Extension

Pointer to the device extension of the device object that represents the parallel port.

Include

parallel.h

Return Value

Number of requests that are queued on the work queue of the parallel port.

Comments

A kernel-mode driver uses an IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO request to obtain the **QueryNumWaiters** pointer.

A driver can use **QueryNumWaiters** to adjust its use of the parallel port based on the number of other clients that are waiting for access to the parallel port. Note that Parport queues only allocate and select requests.

QueryNumWaiters runs in the caller's thread at an IRQL <= DISPATCH_LEVEL.

See Also

FreePort, **TryAllocatePort**

TryAllocatePort

```
BOOLEAN
(*TryAllocatePort)(
  IN PVOID Context
)
```

The **TryAllocatePort** callback routine is a non-blocking routine that a kernel-mode driver can use to allocate a parallel port.

Parameters

Context

Pointer to the extension of the device object that represents the parallel port.

Include

parallel.h

Return Value

TRUE

The port was allocated.

FALSE

The port was not allocated.

Comments

A kernel-mode driver sends an IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO request to obtain the **TryAllocatePort** pointer.

A driver can use **TryAllocatePort** to allocate a parallel port instead of using an IOCTL_INTERNAL_PARALLEL_PORT_ALLOCATE request. **TryAllocatePort** is non-blocking, does not queue a port allocate request, and returns immediately.

*If a client uses only **TryAllocatePort** to attempt to allocate a port for which other clients are contending, Parport might never allocate the port to the client.* To ensure success, a client must use a port allocate request. Parport queues, and subsequently processes, port allocate requests and device select requests in the order in which the requests are received.

TryAllocatePort runs in the caller's thread at an IRQL <= DISPATCH_LEVEL.

See Also

FreePort, **FreePortFromInterruptLevel**, **QueryNumWaiters**, **TryAllocatePortAtInterruptLevel**

TryAllocatePortAtInterruptLevel

```
BOOLEAN
(*TryAllocatePortAtInterruptLevel)(
  IN PVOID Context
)
```

The **TryAllocatePortAtInterruptLevel** callback routine allocates a parallel port at IRQL DIRQL.

Parameters

Context

Pointer to the extension of the device object that represents the parallel port.

Include

parallel.h

Return Value

TRUE

The port was allocated.

FALSE

The port was not allocated.

Comments

A kernel-mode driver sends an IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO request to obtain the **TryAllocatePortAtInterruptLevel** pointer.

TryAllocatePortAtInterruptLevel is non-blocking, does not queue an allocate request, and returns immediately.

A driver uses **TryAllocatePortAtInterruptLevel** in conjunction with an ISR. If the driver does not have a port allocated when the driver's ISR is called, the driver can use **TryAllocatePortAtInterruptLevel**.

See Also

FreePort, **FreePortFromInterruptLevel**, **QueryNumWaiters**, **TryAllocatePort**

TrySelectDevice

```
NTSTATUS
(*TrySelectDevice)(
  IN PVOID Context,
  IN PVOID TrySelectCommand
)
```

The **TrySelectDevice** callback routine selects an IEEE 1284.3 daisy-chain device or an IEEE 1284 end-of-chain device on a parallel port.

Parameters

Context

Pointer to the extension of the device object that represents the parallel port.

TrySelectCommand

Pointer to a PARALLEL_1284_COMMAND structure. The caller specifies the following parameters:

ID

Specifies the 1284.3 device ID.

CommandFlags

Specifies a bitwise OR of zero or more of the following flags:

Value	Description
PAR_END_OF_CHAIN_DEVICE	Specifies an end-of-chain device.
PAR_HAVE_PORT_KEEP_PORT	Specifies that the caller has the port allocated and to keep the port allocated.

Include

parallel.h

Return Value

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

The device ID is not valid.

STATUS_PENDING

The caller did not specify PAR_HAVE_PORT_KEEP_PORT, and the port is already allocated.

STATUS_UNSUCCESSFUL

The caller has allocated the port, but Parport could not select the device.

Comments

A kernel-mode driver uses an IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO request to obtain the **TrySelectDevice** pointer.

A kernel-mode driver can use an IOCTL_INTERNAL_SELECT_DEVICE request or **TrySelectDevice** to select a device on a parallel port. Parport queues a select request if the

parallel port is already allocated. However, **TrySelectDevice** does not queue a select request, and the routine returns immediately if the port cannot be allocated.

If a client uses only **TrySelectDevice** to attempt to select a device, and other clients are contending for the port, Parport might never allocate the port to the client. To ensure success, a client must use a select device request. Parport queues, and subsequently processes, port allocate requests and device select requests in the order in which the requests are received.

TrySelectDevice runs in the caller's thread at IRQL <= DISPATCH_LEVEL.

See Also

PARALLEL_PNP_INFORMATION, IOCTL_INTERNAL_SELECT_DEVICE

TrySetChipMode

```
NTSTATUS
(*TrySetChipMode) (
  IN PDEVICE_EXTENSION Extension,
  IN UCHAR ChipMode
)
```

The **TrySetChipMode** callback routine sets the operating mode of a parallel port.

Members

Extension

Pointer to the extension of the device object that represents the parallel port.

ChipMode

Specifies the operating mode of the parallel port. (For more information on operating modes, see the ECR modes defined in *%Windows 2000 DDK install directory%\inc\ddk\parallel.h.*).

Include

parallel.h

Return Value

STATUS_SUCCESS

STATUS_INVALID_DEVICE_STATE

The mode is not cleared.

STATUS_NO_SUCH_DEVICE

The specified operating mode is not valid.

Comments

A kernel-mode driver uses an IOCTL_INTERNAL_GET_PARALLEL_PORT_INFO request to obtain the **TrySetChipMode** pointer.

The **TrySetChipMode** callback routine sets the operating mode of a parallel port. A caller uses **TrySetChipMode** in conjunction with **ClearChipMode**.

To set a new mode, a caller must first clear the current mode.

To clear the current mode, a caller must specify the same mode that was used to set the current mode.

See Also

ClearChipMode, IOCTL_INTERNAL_PARALLEL_CLEAR_CHIP_MODE, IOCTL_INTERNAL_PARALLEL_SET_CHIP_MODE

CHAPTER 4

Parclass Driver Reference

This chapter describes the following topics about *Parclass*, the Windows® 2000 system class driver for parallel devices that are attached to parallel ports:

- *Parclass Major I/O Requests*
- *Parclass Device Control Requests*
- *Parclass Internal Device Control Requests*
- *Parclass Data types*
- *Parclass Callback Routines*

Parclass creates and administers a bus for all Plug and Play parallel devices that are attached to parallel ports. The Parclass service, which is part of the extended base group of services, depends on the *Parport* service. The executable image of the Parclass is *parallel.sys*.

For more information on Parclass and Parport, see:

- *Parallel Devices and Drivers* in the online DDK
- Sample code in the *%install directory%\src\kernel\parclass* directory in the Windows 2000 DDK
- Sample code in the *%install directory%\src\kernel\parport* directory in the Windows 2000 DDK
- Include files *%install directory%\inc\ddk\parallel.h* and *%install directory%\inc\ntddpar.h* in the Windows 2000 DDK

Parclass Major I/O Requests

This section describes the Parclass-specific handling of the following I/O requests that Parclass supports:

IRP_MJ_CLEANUP
IRP_MJ_CLOSE
IRP_MJ_CREATE
IRP_MJ_DEVICE_CONTROL
IRP_MJ_INTERNAL_DEVICE_CONTROL
IRP_MJ_QUERY_INFORMATION
IRP_MJ_PNP
IRP_MJ_POWER
IRP_MJ_READ
IRP_MJ_SYSTEM_CONTROL
IRP_MJ_WRITE

See the following topics for information about Parclass's generic handling of these I/O requests:

- *IRP Function Codes and IOCTLs*
- Sample Parclass code in the *%install directory%\src\kernel\parclass* directory in the Windows 2000 DDK
- *Plug and Play IRPs* in Volume 1
- *I/O Requests for Power Management* in Volume 1
- *WMI IRPs* in the online DDK

IRP_MJ_CREATE

Operation

The IRP_MJ_CREATE request opens a parallel device. The following Parclass-specific considerations apply to opening a parallel device:

- Parallel devices are exclusive devices. Parclass fails an IRP_MJ_CREATE request if a device is already open.
- A kernel-mode driver that connects to a parallel port or attaches an FDO to a Parclass PDO must open the Parclass PDO before the driver can send read, write, or device control requests to the parallel device.

Status I/O Block

- The **Information** member is set to zero.
- The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_ACCESS_DENIED

The device is already open.

STATUS_DELETE_PENDING

STATUS_DEVICE_REMOVED

STATUS_INVALID_DEVICE_REQUEST

There is no hardware present.

STATUS_NOT_A_DIRECTORY

IRP_MJ_DEVICE_CONTROL

Operation

Parclass supports the following major device control requests:

IOCTL_IEEE1284_GET_MODE
IOCTL_IEEE1284_NEGOTIATE
IOCTL_PAR_GET_DEFAULT_MODES
IOCTL_PAR_GET_DEVICE_CAPS
IOCTL_PAR_QUERY_DEVICE_ID
IOCTL_PAR_QUERY_DEVICE_ID_SIZE
IOCTL_PAR_QUERY_INFORMATION
IOCTL_PAR_QUERY_RAW_DEVICE_ID
IOCTL_PAR_SET_INFORMATION
IOCTL_PAR_SET_READ_ADDRESS
IOCTL_PAR_SET_WRITE_ADDRESS
IOCTL_SERIAL_GET_TIMEOUTS
IOCTL_SERIAL_SET_TIMEOUTS

No other device control requests are supported.

For more information on the device control requests that Parclass supports, see *Parclass Device Control Requests*.

Status I/O Block

The values of the status block members are request-specific. If the request is not supported, the **Status** member is set to STATUS_INVALID_PARAMETER.

IRP_MJ_INTERNAL_DEVICE_CONTROL

Operation

Parclass supports the following internal device control requests:

IOCTL_INTERNAL_DISCONNECT_IDLE
IOCTL_INTERNAL_LOCK_PORT
IOCTL_INTERNAL_PARCLASS_CONNECT
IOCTL_INTERNAL_PARCLASS_DISCONNECT
IOCTL_INTERNAL_PARDOT3_CONNECT
IOCTL_INTERNAL_PARDOT3_DISCONNECT
IOCTL_INTERNAL_UNLOCK_PORT

No other internal device control requests are supported.

For more information, see *Parclass Internal Device Control Requests* in this chapter.

Status I/O Block

The status block values are specific to each request. If an internal device control request is not supported, the **Status** member is set to STATUS_INVALID_PARAMETER.

IRP_MJ_QUERY_INFORMATION

Operation

The IRP_MJ_QUERY_INFORMATION request returns file information for a parallel device.

Parclass supports queries for the following types of information:

- **FileStandardInformation**
- **FilePositionInformation**

Input

The **Parameters.QueryFile.FileInformationClass** member is set to **FileStandard-Information** or **FilePositionInformation**.

FileStandardInformation Request

The **AssociatedIrp.SystemBuffer** member points to a FILE_STANDARD_INFORMATION structure that the client allocates for output of file information.

The **Parameters.QueryFile.Length** member is set to the size in bytes of a FILE_STANDARD_INFORMATION structure.

FilePositionInformation Request

AssociatedIrp.SystemBuffer points to a FILE_POSITION_INFORMATION structure that the client allocates for output of file information.

The **Parameters.SetFile.Length** member is set to the size in bytes of a FILE_POSITION_INFORMATION structure.

Output

AssociatedIrp.SystemBuffer points to the requested information.

FileStandardInformation Request Type

Sets the following members in the FILE_STANDARD_INFORMATION structure:

- **AllocationSize.QuadPart** set to zero.
- **EndOfFile** is set to the value of the **AllocationSize** member.
- **NumberOfLinks** is set to zero.
- **DeletePending** is set to FALSE.
- **Directory** is set to FALSE.

FilePositionInformation Request Type

Sets the **CurrentByteOffset.QuadPart** member of a FILE_POSITION_INFORMATION structure to zero.

Status I/O Block

If the request succeeds, the **Information** member is set to the size in bytes of the structure associated with the type of request. Otherwise, the **Information** member is set to zero.

The Status member is set to one of the following status values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The size in bytes of the structure specified by the input parameter is less than the size in bytes of the structure associated with the request type.

STATUS_DEVICE_REMOVED

STATUS_INVALID_PARAMETER

The specified type of information is not valid.

IRP_MJ_READ

Operation

The IRP_MJ_READ request transfers data from a parallel device to the client. Parclass uses the read protocol set for the parallel device. The default read protocol is NIBBLE_MODE. A client can negotiate a read protocol by using an IOCTL_IEEE1284_NEGOTIATE request.

Parclass sets a cancel routine for the read request, marks the read request as pending, and queues the read request on a work queue. The read request is held on the work queue in a state that can be canceled until the read request is either completed or canceled by the client.

Input

The **Parameters.Read.Length** member points to the number of bytes to read from the parallel device.

Output

The **AssociatedIrp.SystemBuffer** member points to a read buffer that the client allocates for the read data. The buffer must be large enough to hold the requested number of bytes.

Status I/O Block

The **Information** member is set to the number of bytes actually read from the parallel device.

The **Status** member is set to one of the following status values:

STATUS_SUCCESS

STATUS_DELETE_PENDING

STATUS_CANCELLED

STATUS_PENDING

The request is queued on a work queue for the parallel device.

STATUS_INVALID_PARAMETER

The **Parameters.Write.ByteOffset** member is not zero. Note that both read and write requests use this member.

STATUS_DEVICE_REMOVED

IRP_MJ_WRITE

Operation

The IRP_MJ_WRITE request transfers data from the client to the parallel device.

Parclass transfers data from the client to the parallel device by using the write protocol that is set for the parallel device. The default write protocol is CENTRONICS_MODE. A client can negotiate a write protocol by using an IOCTL_IEEE1284_NEGOTIATE request.

Parclass sets a cancel routine for the write request, marks the write request as pending, and queues the write request on a work queue. The write request is held in a state that can be canceled until the request is either completed or canceled.

Input

The **AssociatedIrp.SystemBuffer** points to a write buffer that the client allocates for write data. The buffer must be large enough to hold the requested number of bytes to write to the parallel device.

The **Parameters.Write.Length** member points to the number of bytes to write to the parallel device.

Status I/O Block

The **Information** member is set to the number of bytes actually written to the parallel device.

The **Status** member is set to STATUS_SUCCESS or one or the following values:

STATUS_DELETE_PENDING

STATUS_CANCELLED

STATUS_PENDING

The request is queued on a work queue for the parallel device.

STATUS_INVALID_PARAMETER

The **Parameters.Write.ByteOffset** member is not zero.

STATUS_DEVICE_REMOVED

Parclass Device Control Requests

This section describes the following topics:

IOCTL_IEEE1284_GET_MODE
IOCTL_IEEE1284_NEGOTIATE
IOCTL_PAR_GET_DEFAULT_MODES
IOCTL_PAR_GET_DEVICE_CAPS
IOCTL_PAR_IS_PORT_FREE
IOCTL_PAR_QUERY_DEVICE_ID
IOCTL_PAR_QUERY_DEVICE_ID_SIZE
IOCTL_PAR_QUERY_INFORMATION
IOCTL_PAR_QUERY_RAW_DEVICE_ID
IOCTL_PAR_SET_INFORMATION
IOCTL_PAR_SET_READ_ADDRESS
IOCTL_PAR_SET_WRITE_ADDRESS
IOCTL_SERIAL_GET_TIMEOUTS
IOCTL_SERIAL_SET_TIMEOUTS

No other device control requests are supported by Parclass. Parclass completes unsupported device control requests with a status of STATUS_INVALID_PARAMETER.

IOCTL_IEEE1284_GET_MODE

Operation

The IOCTL_IEEE1284_GET_MODE request returns the IEEE 1284 read and write protocols that are currently set for the parallel device. For information on the communication modes that Parclass supports, see the modes NONE through ECP_ANY that are defined in *%install directory%\inc\ntddpar.h.*

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a PARCLASS_NEGOTIATION_MASK structure.

Output

The **AssociatedIrp.SystemBuffer** member points to a PARCLASS_NEGOTIATION_MASK structure that the client allocates to output mode information. Parclass specifies the read (reverse) protocol in the **usReadMask** member and the write (forward) protocol in the **usWriteMask** member.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of a PARCLASS_NEGOTIATION_MASK. Otherwise, the **Information** member is set to zero.

The **Status** field is set to one of the following status values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of **Parameters.DeviceIoControl.OutputBufferLength** is less than the size in bytes of a PARCLASS_NEGOTIATION_MASK structure.

IOCTL_IEEE1284_NEGOTIATE

Operation

The IOCTL_IEEE1284_NEGOTIATE request sets the read and write protocols that are used for a parallel device. Parclass negotiates with the parallel device to determine the fastest mode that the device supports from among the modes that are specified by the client. Parclass sets the default read and write modes to the negotiated modes. For information on the communication modes that Parclass supports, see the modes NONE through ECP_ANY defined in *%install directory%\inc\ntddpar.h.*

Input

The **AssociatedIrp.SystemBuffer** member points to a PARCLASS_NEGOTIATION_MASK structure that the client allocates for the input and output of mode information. The client sets the **usReadMask** and **usWriteMask** members.

The **Parameters.DeviceIoControl.InputBufferLength** member is set to the size in bytes of a PARCLASS_NEGOTIATION_MASK structure.

Output

The **AssociatedIrp.SystemBuffer** points to the PARCLASS_NEGOTIATION_MASK structure that Parclass uses to output mode information. Parclass sets the **usReadMask** and **usWriteMask** members to the negotiated modes.

I/O Status Block

If request is successful, the **Information** member is set to the size in bytes of a PARCLASS_NEGOTIATION_MASK structure. Otherwise the **Information** field is set to zero.

The **Status** field is set to one of the following status values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

The value of the **Parameters.DeviceIoControl.InputBufferLength** member is less than the size in bytes of a PARCLASS_NEGOTIATION_MASK.

IOCTL_PAR_GET_DEFAULT_MODES

Operation

The IOCTL_PAR_GET_DEFAULT_MODES request returns the default write (forward) and read (reverse) IEEE 1284 protocols. The default write protocol is CENTRONICS_MODE; the default read protocol is NIBBLE. For information on the communication modes that Parclass supports, see the modes NONE through ECP_ANY defined in *%install directory%\inc\ntddpar.h.*

Input

The value of the **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a PARCLASS_NEGOTIATION_MASK structure.

Output

The **AssociatedIrp.SystemBuffer** member points to a PARCLASS_NEGOTIATION_MASK structure that the client allocates to output mode information. Parclass sets the **usReadMask** member and the **usWriteMask** member. The default write mode is CENTRONICS_MODE; the default read mode is NIBBLE.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of a PARCLASS_NEGOTIATION_MASK structure. Otherwise, **Information** is set to zero.

The **Status** field is set to one of the following status values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.OutputBufferLength** is less than the size in bytes of a PARCLASS_NEGOTIATION_MASK structure.

IOCTL_PAR_GET_DEVICE_CAPS

Operation

The IOCTL_PAR_GET_DEVICE_CAPS request does the following:

- Returns the operating protocols that the parallel device supports
- Specifies the protocols that Parclass must not use with a device

Input

The **AssociatedIrp.SystemBuffer** member points to a USHORT buffer that the client allocates to input and output mode information. The request sets the input buffer to a logical OR of the modes that Parclass must not use with a parallel device.

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a USHORT.

Output

AssociatedIrp.SystemBuffer points to the USHORT buffer that Parclass uses to output mode information. Parclass sets the buffer to indicate which operating protocols the parallel device supports.

I/O Status Block

The **Information** field is set to the size in bytes of a USHORT.

The **Status** field is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The **Parameters.DeviceIoControl.OutputBufferLength** field is less than the size in bytes of a USHORT.

IOCTL_PAR_IS_PORT_FREE

Operation

The IOCTL_PAR_IS_PORT_FREE request determines if a parallel port is free at the time Parclass processes the request. This request is provided primarily for user-mode clients.

The request is processed immediately after the I/O Manager calls Parclass's dispatch routine for device control requests. Note, however, that the status of the port can change between

the time that Parclass completes the request and the time that control returns to a user-mode client.

Kernel-mode clients can directly determine if a parallel port is free by calling Parport's **TryAllocatePort** callback routine.

Output

The **AssociatedIrp.SystemBuffer** member points to a BOOLEAN buffer that the client allocates to output the status of the port. If the port is free, Parclass sets the buffer to TRUE, otherwise it sets the buffer to FALSE.

I/O Status Block

The **Information** member is set to the size in bytes of a BOOLEAN.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The **Parameters.DeviceIoControl.OutputBufferLength** member is less than the size in bytes of a BOOLEAN.

IOCTL_PAR_QUERY_DEVICE_ID

Operation

The IOCTL_PAR_QUERY_DEVICE_ID request returns the IEEE 1284 device ID.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member specifies the size in bytes of the output buffer that can hold both the device ID and a NULL terminator. A client can use IOCTL_PAR_QUERY_DEVICE_ID_SIZE to determine the required buffer size. A device ID can be up to 64 KB in size.

Output

The **AssociatedIrp.SystemBuffer** member points to a buffer that the client allocates to output the device ID. The buffer contains the device ID and a NULL terminator.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of a buffer that holds both the device ID and a NULL terminator. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the following status values:

STATUS_SUCCESS

STATUS_IO_DEVICE_ERROR

STATUS_BUFFER_TOO_SMALL

IOCTL_PAR_QUERY_DEVICE_ID_SIZE

Operation

The IOCTL_PAR_QUERY_DEVICE_ID_SIZE returns the size in bytes of a buffer that can hold a device's IEEE 1284 device ID and a NULL terminator.

Output

The **AssociatedIrp.SystemBuffer** member points to a PAR_DEVICE_ID_SIZE_ INFORMATION structure that the client allocates to output the device ID size information. Parclass sets the **DeviceIdSize** member of the output structure to the size in bytes of a buffer that can hold the device ID and a NULL terminator.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of a PAR_ DEVICE_ID_SIZE_INFORMATION structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the following status values:

STATUS_SUCCESS

STATUS_IO_DEVICE_ERROR

IOCTL_PAR_QUERY_INFORMATION

Operation

The IOCTL_PAR_QUERY_INFORMATION request returns the status of an IEEE 1284 end-of-chain device.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a PAR_QUERY_INFORMATION structure.

Output

The **AssociatedIrp.SystemBuffer** member points to a PAR_QUERY_INFORMATION structure that the client allocates to output status information. Parclass sets the **Status** member to a logical OR of one or more of the following operating conditions:

PARALLEL_BUSY
PARALLEL_NOT_CONNECTED
PARALLEL_OFF_LINE
PARALLEL_PAPER_EMPTY
PARALLEL_POWER_OFF
PARALLEL_SELECTED

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a PAR_QUERY_INFORMATION structure. Otherwise, the **Information** is set to zero.

The **Status** member is set to one of the following status values:

STATUS_SUCCESSFUL

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.OutputBufferLength** member is less than the size in bytes of a PAR_QUERY_INFORMATION structure.

STATUS_CANCELLED

STATUS_PENDING

The request is queued on a work queue for the parallel device.

IOCTL_PAR_QUERY_RAW_DEVICE_ID

Operation

The IOCTL_PAR_QUERY_RAW_DEVICE_ID request performs the same operation as the IOCTL_PAR_QUERY_DEVICE_ID request.

IOCTL_PAR_SET_INFORMATION

Operation

The IOCTL_PAR_SET_INFORMATION request resets and initializes a parallel device.

Input

The **AssociatedIrp.SystemBuffer** member points to a PAR_SET_INFORMATION structure that the client allocates to input set information. The request sets the **Init** member to PARALLEL_INIT.

The **Parameters.DeviceIoControl.InputBufferLength** member is set to the size in bytes of a PAR_SET_INFORMATION structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following status values:

STATUS_SUCCESS

STATUS_CANCELLED

STATUS_BUFFER_TOO_SMALL

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of a PAR_SET_INFORMATION structure.

STATUS_DEVICE_NOT_CONNECTED

STATUS_DEVICE_OFF_LINE

STATUS_DEVICE_PAPER_EMPTY

STATUS_DEVICE_POWERED_OFF

STATUS_INVALID_PARAMETER

The request does not specify PARALLEL_INIT.

STATUS_PENDING

The request is queued on a work queue for the parallel device.

IOCTL_PAR_SET_READ_ADDRESS

Operation

The IOCTL_PAR_SET_READ_ADDRESS request sets an ECP or EPP read address (channel) for a parallel device. Parclass queues this request on a work queue for the parallel device.

Input

The **AssociatedIrp.SystemBuffer** member points to a UCHAR buffer that the client allocates to input a read address. The request sets the buffer to an ECP or EPP read address.

Parameters.DeviceIoControl.InputBufferLength member is set to the size in bytes of a UCHAR.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_CANCELLED

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of a UCHAR.

STATUS_PENDING

The request is queued on a work queue for the parallel device.

IOCTL_PAR_SET_WRITE_ADDRESS

The IOCTL_PAR_SET_WRITE_ADDRESS request sets an ECP or EPP write address (channel) for the parallel device. Parclass queues this request on a work queue for the parallel device.

Input

The **AssociatedIrp.SystemBuffer** member points to a UCHAR buffer that the client allocates to inputa write address. The client sets the buffer to an ECP or EPP write address.

Parameters.DeviceIoControl.InputBufferLength member is set to the size in bytes of a UCHAR.

Status I/O Block

The **Information** field is set to zero.

The **Status** field is set to one of the following values:

STATUS_SUCCESS

STATUS_CANCELLED

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of a UCHAR.

STATUS_PENDING

The request is queued on a work queue for the parallel device.

STATUS_UNSUCCESSFUL

IOCTL_SERIAL_GET_TIMEOUTS

Operation

The IOCTL_SERIAL_GET_TIMEOUTS request returns the current setting of the timeout value that Parclass uses with write requests. Parclass does not queue a get timeouts request. The write timeout value is used with SPP and SW_ECP modes.

A client uses an IOCTL_SERIAL_SET_TIMEOUTS request to set timeouts.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** field is set to the size in bytes of a SERIAL_TIMEOUTS structure.

Output

The **AssociatedIrp.SystemBuffer** points to a SERIAL_TIMEOUTS structure that the client allocates to output timeout information. Parclass sets the **WriteTotalTimeoutConstant** member to the timeout value in milliseconds.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a SERIAL_TIMEOUTS structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.OutputBufferLength** member is less than the size in bytes of a SERIAL_TIMEOUTS structure.

IOCTL_SERIAL_SET_TIMEOUTS

Operation

An IOCTL_SERIAL_SET_TIMEOUTS request resets the timeout value that Parclass uses with write requests. The write timeout value is used with SPP and SW_ECP modes. Parclass queues a set timeout request on a work queue for the parallel device.

A client uses an IOCTL_SERIAL_GET_TIMEOUTS to obtain the timeout values.

Input

The **AssociatedIrp.SystemBuffer** points to a SERIAL_TIMEOUTS structure that the client allocates to input timeout information. The client sets the **WriteTotalTimeoutConstant** member to a value in milliseconds.

The **Parameters.DeviceIoControl.OutputBufferLength** field is set to the size in bytes of a SERIAL_TIMEOUTS structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.InputBufferLength** member is less than the size in bytes of a SERIAL_TIMEOUTS structure.

STATUS_INVALID_PARAMETER

The requested timeout value is less than two seconds.

STATUS_PENDING

The request is queued on a work queue for the parallel device.

Parclass Internal Device Control Requests

This section describes the following topics:

IOCTL_INTERNAL_DISCONNECT_IDLE
IOCTL_INTERNAL_LOCK_PORT
IOCTL_INTERNAL_PARCLASS_CONNECT
IOCTL_INTERNAL_PARCLASS_DISCONNECT
IOCTL_INTERNAL_PARDOT3_CONNECT
IOCTL_INTERNAL_PARDOT3_DISCONNECT
IOCTL_INTERNAL_UNLOCK_PORT

No other internal device control requests are supported by Parclass. Parclass completes unsupported internal device control requests with a status of STATUS_INVALID_PARAMETER.

IOCTL_INTERNAL_DISCONNECT_IDLE

Operation

The IOCTL_INTERNAL_DISCONNECT_IDLE request disconnects the IEEE 1284 operating modes that are set for a parallel device. Parclass sets the default operating mode to IEEE 1284-compatible.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_DELETE_PENDING

STATUS_DEVICE_REMOVED

STATUS_PENDING

The request is queued on a work queue for the parallel device.

IOCTL_INTERNAL_LOCK_PORT

Operation

The IOCTL_INTERNAL_LOCK_PORT request *locks* a parallel port for the exclusive use of a parallel device. This request also selects the parallel device. If a client has locked the port, Parclass does not automatically free the port just before completing requests that require that the parallel port to be allocated. A client must unlock the port to permit other clients to access parallel devices on the parallel port.

Status I/O Block

The **Information** field is set to zero.

The **Status** field is set to one of the following values:

STATUS_SUCCESS

STATUS_DELETE_PENDING

STATUS_DEVICE_REMOVED

STATUS_PENDING

The request is queued on a work queue for the parallel device.

STATUS_UNSUCCESSFUL

IOCTL_INTERNAL_PARCLASS_CONNECT

Operation

The IOCTL_INTERNAL_PARCLASS_CONNECT request returns information about the parallel port and the callback routines that Parclass provides. Typically, a client first uses a connect request to obtain connect information, and then uses a lock port request to allocate exclusive use of the parallel port for a parallel device. Parclass does not queue this request.

Input

The value of the **Parameters.DeviceIoControl.OutputBufferLength** member is set to the size in bytes of a PARCLASS_INFORMATION structure.

Output

The **AssociatedIrp.SystemBuffer** member points to a PARCLASS_INFORMATION structure that the client allocates to output Parclass information.

Status I/O Block

If the request is successful, the **Information** member is set to the size in bytes of a PARCLASS_INFORMATION structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of the **Parameters.DeviceIoControl.OutputBufferLength** member is less than the size in bytes of a PARCLASS_INFORMATION structure.

STATUS_DELETE_PENDING

STATUS_DEVICE_REMOVED

IOCTL_INTERNAL_PARCLASS_DISCONNECT

Operation

The IOCTL_INTERNAL_PARCLASS_DISCONNECT request disconnects a client from a parallel device.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following status values:

STATUS_SUCCESS

STATUS_DELETE_PENDING

STATUS_DEVICE_REMOVED

IOCTL_INTERNAL_PARDOT3_CONNECT

Operation

The IOCTL_INTERNAL_PARDOT3_CONNECT request supports the IEEE 1284.3 connect datalink service defined in *IEEE P1284.3, Draft 4.91, May 27, 1998.*

The operation of this request will be described in a future release of the Windows 2000 DDK.

IOCTL_INTERNAL_PARDOT3_DISCONNECT

Operation

The IOCTL_INTERNAL_PARDOT3_DISCONNECT request supports the IEEE 1284.3 disconnect data link service defined in *IEEE P1284.3, Draft 4.91, May 27, 1998.*

The operation of this request will be described in a future release of the Windows 2000 DDK.

IOCTL_INTERNAL_UNLOCK_PORT

Operation

The IOCTL_INTERNAL_LOCK_PORT request unlocks a parallel port that was locked by an IOCTL_INTERNAL_LOCK_PORT request.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_DELETE_PENDING

STATUS_DEVICE_REMOVED

STATUS_PENDING

The request is queued on a work queue for the parallel device.

Parclass Data Types

This section describes the following data types that Parclass uses to input and output information:

PAR_QUERY_INFORMATION

Specifies the operating status of the parallel port.

PAR_SET_INFORMATION

Specifies an initial operating status for the parallel port.

PARCLASS_INFORMATION

Specifies information about the parallel port, callback routines to operate the parallel port, and callbacks to read and write a parallel device.

PARCLASS_NEGOTIATION_MASK

Specifies the read and write protocols that a client selects for a parallel device.

PAR_QUERY_INFORMATION

```
typedef struct _PAR_QUERY_INFORMATION{
    UCHAR Status;
} PAR_QUERY_INFORMATION, *PPAR_QUERY_INFORMATION;
```

The PAR_QUERY_INFORMATION structure specifies the operating status of a parallel port.

Members

Status

Specifies the operating status of a parallel port. The value of **Status** is a logical OR of one or more of the following values:

PARALLEL_INIT

PARALLEL_AUTOFEED

PARALLEL_PAPER_EMPTY

PARALLEL_OFF_LINE

PARALLEL_POWER_OFF

PARALLEL_NOT_CONNECTED

PARALLEL_BUSY

PARALLEL_SELECTED

Include

ntddpar.h

Comments

This structure is used with an IOCTL_PAR_QUERY_INFORMATION request.

See Also

IOCTL_PAR_QUERY_INFORMATION, IOCTL_PAR_SET_INFORMATION, PAR_SET_INFORMATION

PAR_SET_INFORMATION

```
typedef struct _PAR_SET_INFORMATION{
        UCHAR Init;
} PAR_SET_INFORMATION, *PPAR_SET_INFORMATION;
```

The PAR_SET_INFORMATION structure specifies the initial operating status of a parallel port.

Members

Init

Specifies the operating status of the parallel port. Must be set to PARALLEL_INIT.

Include

ntddpar.h

Comments

This structure is used with an IOCTL_PAR_SET_INFORMATION request.

See Also

IOCTL_PAR_QUERY_INFORMATION, IOCTL_PAR_SET_INFORMATION, PAR_QUERY_INFORMATION

PARCLASS_INFORMATION

```
typedef struct _PARCLASS_INFORMATION {
    PUCHAR                      Controller;
    ULONG                       SpanOfController;
    PDETERMINE_IEEE_MODES       DetermineIeeeModes;
    PNEGOTIATE_IEEE_MODE        NegotiateIeeeMode;
    PTERMINATE_IEEE_MODE        TerminateIeeeMode;
    PPARALLEL_IEEE_FWD_TO_REV   IeeeFwdToRevMode;
    PPARALLEL_IEEE_REV_TO_FWD   IeeeRevToFwdMode;
    PPARALLEL_READ              ParallelRead;
    PPARALLEL_WRITE             ParallelWrite;
    PVOID                       ParclassContext;
    ULONG                       HardwareCapabilities;
    ULONG                       FifoDepth;
    ULONG                       FifoWidth;
} PARCLASS_INFORMATION, *PPARCLASS_INFORMATION;
```

The PARCLASS_INFORMATON structure contains information about a parallel port, pointers to callback routines to operate a parallel port, and pointers to callback routines to read and write to a parallel device.

Members

Controller

Specifies the base I/O address allocated to a parallel port.

SpanOfController

Specifies the range in bytes of I/O address space allocated to a parallel port.

DetermineIeeeModes

Pointer to a callback routine that determines which IEEE protocols a parallel device supports.

NegotiateIeeeMode

Pointer to a callback routine that negotiates the fastest protocol that Parclass supports from among those specified by the caller.

TerminateIeeeMode

Pointer to a callback routine that terminates the current IEEE mode and sets the mode to IEEE_COMPATIBILITY.

IeeeFwdToRevMode

Pointer to a callback routine that changes the transfer mode from forward to reverse.

IeeeRevToFwdMode

Pointer to a callback routine that changes the transfer mode from reverse to forward.

ParallelRead

Pointer to a callback routine that a client can use to read from a parallel device.

ParallelWrite

Pointer to a callback routine that a client can use to write to a parallel device.

ParclassContext

Pointer to the device extension of a parallel device object.

HardwareCapabilities

Specifies which hardware capabilities are present. **HardwareCapabilities** is a logical OR of one or more of the following values:

PPT_NO_HARDWARE_PRESENT

PPT_ECP_PRESENT

PPT_EPP_PRESENT

PPT_EPP_32_PRESENT

32-bit reads and writes are supported.

PPT_BYTE_PRESENT

PPT_BIDI_PRESENT

PPT_1284_3_PRESENT

FifoDepth

Specifies the size in bytes of the ECP FIFO.

FifoWidth

Species the width in bits of the ECP FIFO.

Include

parallel.h

Comments

An upper-level kernel-mode driver can obtain this information from Parclass using an IOCTL_INTERNAL_PARCLASS_CONNECT request. A driver uses this information to

operate a parallel port and to read and write a parallel device. The callback routines can only be used by a driver that holds a lock on the parallel port. A driver obtains a lock by using an IOCTL_INTERNAL_LOCK_PORT request and releases the lock by using IOCTL_INTERNAL_UNLOCK_PORT.

PARCLASS_NEGOTIATION_MASK

```
typedef struct _PARCLASS_NEGOTIATION_MASK {
    USHORT      usReadMask;
    USHORT      usWriteMask;
} PARCLASS_NEGOTIATION_MASK, *PPARCLASS_NEGOTIATION_MASK;
```

The PARCLASS_NEGOTIATION_MASK structure specifies the read and write protocols that a driver selects for a parallel device.

Members

usReadMask

Specifies the read protocols.

usWriteMask

Specifies the write protocols.

Include

ntddpar.h

Comments

A client specifies a set of requested protocols by setting a logical OR of the constants that represent each protocol. Parclass selects the fastest protocol that it supports from among those specified by the client. See the operating modes NONE through ECP_ANY that are defined in *%install directory%\inc\ntddpar.h.*

See Also

IOCTL_IEEE1284_GET_MODE, IOCTL_IEEE1284_NEGOTIATE

Parclass Callback Routines

This section describes the following Parclass callback routines that an upper-level kernel-mode driver can use to operate a parallel-port:

DetermineIeeeModes

Determines which IEEE protocols a parallel device supports.

IeeeFwdToRevMode

Changes the transfer mode from forward to reverse.

IeeeRevToFwdMode

Changes the transfer mode from reverse to forward.

NegotiateIeeeMode

Selects the fastest forward and reverse protocol that Parclass supports from among those specified by the caller. **NegotiateIeeeMode** also connects the transfer mode specified by the caller.

ParallelRead

Reads data from a parallel device.

ParallelWrite

Writes data to a parallel device.

TerminateIeeeMode

Terminates the current IEEE operating mode and sets the mode to IEEE_COMPATIBILITY.

DetermineIeeeModes

```
USHORT
(*DetermineIeeeModes) (
  IN PDEVICE_EXTENSION Extension
)
```

The **DetermineIeeeModes** callback routine determines which IEEE protocols a parallel device supports.

Parameters

Extension

Pointer to a extension of a device object that represents a parallel device.

Include

funcdecl.h

Return Value

The return value indicates which protocols a parallel device supports. The return value is a logical OR of one or more of the following constants:

BOUNDED_ECP

ECP_HW_NOIRQ

EPP_HW

EPP_SW

ECP_SW

IEEE_COMPATIBILITY

CENTRONICS

NONE

These constants represent the protocols that Parclass supports. The protocols are listed in order of decreasing data transfer rate.

Comments

The **DetermineIeeeModes** callback routine runs in the caller's thread at the IRQL of the caller.

See Also

IeeeRevToFwdMode, **IeeeRevToFwdMode**, **NegotiateIeeeMode**, **TerminateIeeeMode**

IeeeFwdToRevMode

```
NTSTATUS
(*IeeeFwdToRevMode)(
  IN PDEVICE_EXTENSION Extension
)
```

The **IeeeFwdToRevMode** callback routine changes the transfer mode from forward to reverse.

Parameters

Extension

Pointer to an extension of a device object that represents a parallel device.

Include

funcdecl.h

Return Value

STATUS_SUCCESS

STATUS_Xxx

An internal operation resulted in an NTSTATUS error.

Comments

If the device is connected and in the reverse mode, the **IeeeFwdToRevMode** callback routine returns without further processing. Otherwise, **IeeeFwdToRevMode** puts the device into reverse mode and connects a previously negotiated reverse protocol. The **NegotiateIeeeMode** callback routine can be used to negotiate the reverse protocol.

IeeeFwdToRevMode runs in the caller's thread at the IRQL of the caller.

See Also

DetermineIeeeModes, **IeeeRevToFwdMode**, **NegotiateIeeeMode**, **TerminateIeeeMode**

IeeeRevToFwdMode

```
NTSTATUS
(*IeeeRevToFwdMode)(
  IN PDEVICE_EXTENSION Extension
)
```

The **IeeeRevToFwdMode** callback routine changes the transfer mode from reverse to forward.

Parameters

Extension

Pointer to an extension of a device object that represents a parallel device.

Include

funcdecl.h

Return Value

STATUS_SUCCESS

STATUS_Xxx

An internal operation resulted in an NTSTATUS error.

Comments

If the device is connected and in the forward mode, the **IeeeRevToFwdMode** callback routine returns without further processing. Otherwise, **IeeeRevToFwdMode** puts a device in the forward mode and connects a previously negotiated forward protocol. The **NegotiateIeeeMode** callback routine can be used to negotiate a forward protocol.

IeeeRevToFwdMode runs in the caller's thread at the IRQL of the caller.

See Also

DetermineIeeeModes, **IeeeFwdToRevMode**, **NegotiateIeeeMode**, **TerminateIeeeMode**

NegotiateIeeeMode

```
NTSTATUS
(*NegotiateIeeeMode)(
  IN PDEVICE_EXTENSION Extension,
  IN USHORT ModeMaskFwd,
  IN USHORT ModeMaskRev,
  IN PARALLEL_SAFETY ModeSafety,
  IN BOOLEAN IsForward
)
```

The **NegotiateIeeeMode** callback routine selects the fastest forward and reverse protocols that Parclass supports from among those specified by the caller. **NegotiateIeeeMode** also connects the transfer mode specified by the caller.

Parameters

Extension

Pointer to the extension of a device object that represents a parallel device.

ModeMaskFwd

Specifies the forward protocols. *ModeMaskFwd* is a logical OR of the constants which represent the protocols that Parclass supports.

ModeMaskRev

Specifies the reverse protocols. *ModeMaskRev* is a logical OR of the constants which represent the protocols that Parclass supports.

ModeSafety

Specifies the safety mode. Must be set to SAFE_MODE.

IsForward

Specifies whether to connect the forward or the reverse protocol that the routine negotiates. If *IsForward* is TRUE, the forward protocol is connected. Otherwise, the reverse protocol is connected.

Include

funcdecl.h

Return Value

STATUS_SUCCESSFUL

STATUS_DEVICE_PROTOCOL_ERROR

An IEEE mode is already set on the device.

STATUS_Xxx

An internal operation resulted in an NTSTATUS error.

Comments

The **NegotiateIeeeMode** callback routine selects the fastest forward and reverse protocols that Parclass supports from among those specified in the forward and reverse mode masks. **NegotiateIeeeMode** also connects the transfer mode specified by the caller.

NegotiateIeeeMode runs in the caller's thread at the IRQL of the caller.

See Also

DetermineIeeeModes, **IeeeFwdToRevMode**, **IeeeRevToFwdMode**, **TerminateIeeeMode**

ParallelRead

```
NTSTATUS
(*ParallelRead)(
  IN PDEVICE_EXTENSION Extension,
  IN PVOID Buffer,

IN ULONG NumBytesToRead,
  OUT PULONG NumBytesRead,
  IN UCHAR Channel
)
```

The **ParallelRead** callback routine reads data from a parallel device.

Parameters

Extension

Pointer to the extension of a device object that represents a parallel device.

Buffer

Pointer to a read buffer that the caller allocates.

NumBytesToRead

Specifies the number of bytes to read.

NumBytesRead

Specifies the number of bytes that were actually copied from the parallel device to the caller's read buffer.

Channel

Not used.

Include

funcdecl.h

Return Value

STATUS_SUCCESS

STATUS_Xxx

An internal operation resulted in an NTSTATUS error.

Comments

A client can use the **ParallelRead** callback routine to read from a parallel device. A client can only use this routine if it has a lock on a parallel port. A client obtains a lock on a parallel port by using an IOCTL_INTERNAL_LOCK_PORT request.

ParallelRead runs in the caller's thread at the IRQL of the caller.

See Also

ParallelWrite

ParallelWrite

```
NTSTATUS
(*ParallelWrite)(
  IN PDEVICE_EXTENSION Extension,
  OUT PVOID Buffer,
  IN ULONG NumBytesToWrite,
  OUT PULONG NumBytesWritten,
  IN UCHAR Channel
)
```

The **ParallelWrite** callback routine writes data to a parallel device.

Parameters

Extension

Pointer to the extension of a device object that represents a parallel device.

Buffer

Pointer to a write buffer that the caller allocates.

NumBytesToWrite

Specifies the number of bytes to copy from the write buffer to the parallel device.

NumBytesWritten

Specifies the number of bytes that were actually copied from the caller's write buffer to the parallel device.

Channel

Not used.

Include

funcdecl.h

Return Value

STATUS_SUCCESS

STATUS_Xxx

An internal operation resulted in an NTSTATUS error.

Comments

A client can use the **ParallelWrite** callback routine to write to a parallel device. A client can only use this routine if it has a lock on a parallel port. A client obtains a lock on a parallel port by using an IOCTL_INTERNAL_LOCK_PORT request.

ParallelWrite runs in the caller's thread at the IRQL of the caller.

See Also

ParallelRead

TerminateIeeeMode

```
NTSTATUS
(*TerminateIeeeMode)(
  IN PDEVICE_EXTENSION Extension
)
```

The **TerminateIeeeModes** callback routine terminates the current IEEE operating mode and sets the mode to IEEE_COMPATIBILITY.

Parameters

Extension

Pointer to the extension of the parallel device object.

Include

funcdecl.h

Return Value

STATUS_SUCCESS

Comments

The **TerminateIeeeModes** callback routine runs in the caller's thread at the IRQL of the caller.

See Also

DetermineIeeeModes, **NegotiateIeeeMode**

PART 3

Drivers for Input Devices

CHAPTER 1

HID I/O Requests

This chapter describes I/O control codes serviced by the Microsoft-supplied HID class driver and by the vendor-supplied HID minidrivers.

I/O Requests Serviced by HID Class Driver

Although user applications can communicate with the HID class driver using the API exposed by *hid.dll*, kernel-mode clients must send device control IRPs directly to the class driver. The following section lists the IOCTL codes that the HID class driver recognizes.

Kernel-mode clients build a device control IRP by calling **IoBuildDeviceIoControlRequest** with arguments that specify the I/O control code and the device object of the top-level collection.

IoBuildDeviceIoControlRequest stores the device object for the top-level collection in the **DeviceObject** member of the current I/O stack location of the IRP.

The **Irp.IoStatus** block is handled differently by each IOCTL. The **Status** member of **Irp.IoStatus** is set by the HID class driver in every case, but the **Information** member might either be set by the class driver or in the lower-level drivers. When data is transferred between the drivers and the hardware device, the lower level drivers record the number of bytes transferred in **Information**, but this value might be overridden by the class driver. See the **IoStatus** section under each IOCTL for a description of how the value of this field is determined for that IOCTL.

IOCTL_HID_GET_POLL_FREQUENCY_MSEC

Operation

Gets the current polling frequency of a top level collection.

Input

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer, which must be >= **sizeof**(ULONG).

Output

Irp->AssociatedIrp.SystemBuffer points to a buffer that will receive the polling frequency.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to **sizeof**(ULONG) if the polling frequency is successfully retrieved.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_SET_POLL_FREQUENCY_MSEC

Operation

Sets the polling frequency of a top level collection.

Clients that do irregular, opportunistic reads on the polled device must furnish a polling interval of zero. In such cases, IOCTL_HID_SET_POLL_FREQUENCY_MSEC does not actually change the polling frequency of the device, but if the report data is not stale when the client does a read, the read is completed immediately with the latest report data for the indicated collection. If the report data is stale, it is refreshed immediately, without waiting for the expiration of the polling interval, and the read is completed with the new data.

If the value for the polling interval provided in the IRP is not zero, then it must be >= MIN_POLL_INTERVAL_MSEC and <= MAX_POLL_INTERVAL_MSEC.

Input

Parameters.DeviceIoControl.InputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the input buffer, which must be >= **sizeof**(ULONG).

Irp->AssociatedIrp.SystemBuffer contains the new polling interval.

Output

None.

I/O Status Block

The HID class driver sets the following field of **Irp.IoStatus**. **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_GET_NUM_DEVICE_INPUT_BUFFERS

Operation

Retrieves the size of the report input queue for a top level collection.

The report input queue is implemented as a ring buffer. Therefore, if the underlying HID device transmits data to the linked HID class/miniclass drivers faster than the client can retrieve the HID reports, device data can be lost. The size of the report input queue can be tuned using IOCTL_SET_NUM_DEVICE_INPUT_BUFFERS.

Input

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer, which must be >= **sizeof**(ULONG).

Output

Irp->AssociatedIrp.SystemBuffer points to a buffer that will receive the size of the report input queue. The size of the buffer is **sizeof**(ULONG).

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to **sizeof**(ULONG) if the size of the report input queue is successfully retrieved.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_SET_NUM_DEVICE_INPUT_BUFFERS

Operation

Sets the size of the report input queue for a top level collection.

The report input queue is implemented as a ring buffer. Therefore, if the underlying HID device transmits data to the linked HID class-miniclass drivers faster than the client can retrieve the HID reports, device data can be lost. The size of the report input queue can be tuned using IOCTL_SET_NUM_DEVICE_INPUT_BUFFERS.

Input

Parameters.DeviceIoControl.InputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the input buffer, which must be >= **sizeof**(ULONG).

Irp->AssociatedIrp.SystemBuffer points to a buffer that will receive the new size of the report input queue. The size of the buffer is **sizeof**(ULONG).

Output

None.

I/O Status Block

The HID class driver sets the following field of **Irp.IoStatus**. **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_GET_COLLECTION_INFORMATION

Operation

Retrieves the collection information for a top level collection, where "collection information" is defined to be the general properties of a collection contained in the HID_COLLECTION_INFORMATION structure.

Among other uses, the information provided in HID_COLLECTION_INFORMATION includes the required buffer size for the collection descriptor in its **DescriptorSize** member. Drivers must provide a buffer of this size when retrieving the collection descriptor with IOCTL_HID_GET_COLLECTION_DESCRIPTOR.

Input

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer, which must be >= **sizeof**(HID_COLLECTION_INFORMATION).

Output

Irp->AssociatedIrp.SystemBuffer points to a buffer that will receive the collection information. This data will be formatted in the client-supplied buffer as a HID_COLLECTION_INFORMATION structure.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to **sizeof**(HID_COLLECTION_INFORMATION) if the data was retrieved successfully.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_GET_COLLECTION_DESCRIPTOR

Operation

Retrieves the collection descriptor for a top level collection.

The collection descriptor contains the preparsed data for this collection that was extracted from the HID device's report descriptor when the device was started.

Kernel-mode clients must allocate a buffer from nonpaged pool of size large enough to hold the collection descriptor. The size of the descriptor can be obtained using IOCTL_HID_GET_COLLECTION_INFORMATION.

Input

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer.

Output

Irp->UserBuffer points to a buffer that will receive the collection descriptor. The collection descriptor will be formatted in the client-supplied buffer as a HIDP_COLLECTION_DESC structure.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes of preparsed data successfully retrieved. If client-supplied buffer was not large enough to store the preparsed data, then **Information** is set to the size in bytes of the buffer required to hold all of the preparsed data.
- **Status** is set to STATUS_SUCCESS if the preparsed data was retrieved without error. Otherwise, it is set to an appropriate NTSTATUS error code. If the client-supplied output buffer is not large enough to hold the preparsed data, then status is set to STATUS_INVALID_BUFFER_SIZE.

IOCTL_HID_FLUSH_QUEUE

Operation

Dequeues all of the unparsed input reports from a top level collection's report input queue.

Input

None.

Output

None.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_GET_FEATURE

Operation

Retrieves a feature report for a top level collection.

Input

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer, which must be >= **sizeof**(HID_XFER_PACKET).

Output

Irp->MdlAddress points to the buffer that will receive the feature report.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_SET_FEATURE

Operation

Sets a feature report for a top level collection.

Input

Parameters.DeviceIoControl.InputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer, which must be >= **sizeof**(HID_XFER_PACKET).

Output

Irp->SystemBuffer points to the buffer that contains the feature report.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_GET_PHYSICAL_DESCRIPTOR

Operation

Requests that the HID device associated with a top level collection provide information about which physical body part is used to control the device.

Input

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer.

Output

Irp->MdlAddress must point to the buffer that will receive the physical descriptor.

I/O Status Block

The HID class driver sets the following field of **Irp.IoStatus**. **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_GET_HARDWARE_ID

Operation

Requests that the HID class driver retrieve the hardware ID from the registry for the HID device associated with a top level collection.

Input

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer.

Output

Irp->MdlAddress points to a buffer to receive the number of device input buffers.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes of registry information retrieved when the IOCTL succeeds.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_GET_MANUFACTURER_STRING

Operation

Requests that the HID class driver instruct the minidriver to send a Get String Descriptor request to the device associated with a top level collection, in order to retrieve the human-readable (or "friendly") name for the device's manufacturer.

Input

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer.

Output

Irp->MdlAddress points to a buffer to receive the manufacturer ID.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_GET_PRODUCT_STRING

Operation

Requests that the HID class driver instruct the minidriver to send a Get String Descriptor request to the device associated with a top level collection, in order to retrieve the human-readable (or "friendly") name for the device's product ID.

Input

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer.

Output

Irp->MdlAddress points to a buffer to receive the product ID string.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_GET_SERIALNUMBER_STRING

Operation

Requests that the HID class driver instruct the minidriver to send a Get String Descriptor request to the device associated with a top level collection, in order to retrieve the human-readable (or "friendly") name for the device's serial number.

Input

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer.

Output

Irp->MdlAddress points to a buffer to receive the serial number string.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_GET_INDEXED_STRING

Operation

Requests that the HID class driver instruct the minidriver to send a Get String Descriptor request to the device associated with a top level collection, in order to retrieve the human-readable (or "friendly") string at the indicated index in the device's string descriptor.

Input

Parameters.DeviceIoControl.InputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the input buffer at the location pointed to by **Irp->AssociatedIrp.SystemBuffer**. The input buffer must be >= **sizeof**(ULONG) and it should contain the index of the string to be retrieved.

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer.

Output

Irp->MdlAddress points to a buffer to receive the retrieved string.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

I/O Requests Serviced by HID Minidrivers

HID minidrivers support only one IRP major function code, IRP_MJ_INTERNAL_DEVICE_CONTROL. The HID class driver and the HID minidriver communicate through IRP_MJ_INTERNAL_DEVICE_CONTROL IOCTLs.

Only the HID class driver sends I/O requests to the HID minidriver. Other drivers communicate with the device through the interface presented by the HID class driver.

HID minidrivers must support each IOCTL documented below.

IOCTL_GET_PHYSICAL_DESCRIPTOR

Operation

Requests the device provide information about which physical body part is used to control the HID device.

Input

Parameters.DeviceIoControl.InputBufferLength is set to the length of the system-resident buffer at **Irp->UserBuffer**.

Output

Miniclass drivers that complete the request to get the physical descriptor copy the results into the user buffer at **Irp->UserBuffer** with the physical descriptor.

I/O Status Block

Miniclass drivers that carry out the I/O to the device set the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

Miniclass drivers that call other drivers with this IRP to carry out the I/O to their device should ensure that the **Information** field of the status block is correct and not alter the contents of the **Status** field.

IOCTL_HID_ACTIVATE_DEVICE

Operation

Makes the device ready for I/O operations.

Input

Parameters.DeviceIoControl.Type3InputBuffer contains the collection identifier, as a ULONG value, of the collection to be made ready.

Output

None.

I/O Status Block

Miniclass drivers that carry out the I/O to the device set the following fields of **Irp.IoStatus**:

- **Information** is set to zero.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

Miniclass drivers that call other drivers with this IRP to carry out the I/O to their device should ensure that the **Information** field of the status block is zero and not alter the contents of the **Status** field.

IOCTL_HID_DEACTIVATE_DEVICE

Operation

Causes the device to cease operations and terminate all outstanding I/O requests.

Input

Parameters.DeviceIoControl.Type3InputBuffer contains the collection identifier, as a ULONG value, of the collection that is ceasing operations.

Output

None.

I/O Status Block

Miniclass drivers that carry out the I/O to the device set the following fields of **Irp.IoStatus**:

- **Information** is set to zero.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

Miniclass drivers that call other drivers with this IRP to carry out the I/O to their device should ensure that the **Information** field of the status block is zero and must not alter the contents of the **Status** field.

IOCTL_HID_GET_DEVICE_ATTRIBUTES

Operation

Retrieves information for the system-supplied class driver for a HID device in a system-defined format.

Input

Parameters.DeviceIoControl.InputBufferLength contains the length, in bytes, of the system-resident buffer at **Irp->UserBuffer**. The buffer supplied is in the form of a HID_DEVICE_ATTRIBUTES structure.

Output

Before completing the request, the miniclass driver must fill in the following information in the HID_DEVICE_ATTRIBUTES structure at **Irp->UserBuffer**:

I/O Status Block

IOCTL_HID_GET_DEVICE_DESCRIPTOR

Operation

Retrieves the device's HID descriptor.

Input

Parameters.DeviceIoControl.OutputBufferLength contains the length of the system-resident buffer provided at **Irp->UserBuffer**.

Output

The miniclass driver fills in the class driver-supplied system-resident buffer provided at **Irp->UserBuffer** with the device descriptor.

I/O Status Block

Miniclass drivers that carry out the I/O to the device set the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

Miniclass drivers that call other drivers with this IRP to carry out the I/O to their device should ensure that the **Information** field of the status block is correct and not alter the contents of the **Status** field.

IOCTL_HID_GET_FEATURE

Operation

Obtains a feature report from the device.

Input

Irp->UserBuffer points to a HID_XFER_PACKET structure that contains the parameters and pointer to a buffer for obtaining the feature report. The following members are used:

- **reportBuffer** points to a resident buffer that the miniclass driver uses to return the feature packet.
- **reportBufferLen** contains the length of the buffer provided at **reportBuffer**.
- **reportId** contains the report identifier, for this collection, of the feature report to be retrieved.

Output

Miniclass drivers fill in the **Irp->UserBuffer->reportBuffer** with the feature report obtained from the device.

I/O Status Block

Miniclass drivers that carry out the I/O to the device set the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

Miniclass drivers that call other drivers with this IRP to carry out the I/O to their device should ensure that the **Information** field of the status block is correct and not alter the contents of the **Status** field.

IOCTL_HID_GET_INDEXED_STRING

Operation

Requests that the HID minidriver retrieve a particular human-readable string from the string descriptor of the device. The minidriver must send a Get String Descriptor request to the device, in order to retrieve the string descriptor, then it must extract the string at the indicated index from the string descriptor and return it in the output buffer indicated by the IRP.

Input

IOCTL_HID_GET_INDEXED_STRING uses two input buffers.

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the output buffer at the location pointed to by **Irp->MdlAddress**.

Parameters.DeviceIoControl.Type3InputBuffer in the I/O stack location of the IRP contains the language ID of the string to be retrieved in its most significant two bytes and the string index in its least significant two bytes.

Output

Irp->MdlAddress points to a buffer to receive the retrieved string. Note that unlike most device control IRPs for HID minidrivers, this IRP does not use METHOD_NEITHER buffering. In particular, it must be distinguished from IOCTL_HID_GET_STRING whose output buffer is identified by **Irp->UserBuffer**.

I/O Status Block

The HID class driver sets the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

IOCTL_HID_GET_REPORT_DESCRIPTOR

Operation

Obtains the report descriptor for the HID device.

Input

Parameters.DeviceIoControl.OutputBufferLength specifies the length, in bytes, of the locked-down buffer at **Irp->UserBuffer**.

Output

The miniclass driver fills the buffer at **Irp->UserBuffer** with the report descriptor.

I/O Status Block

Miniclass drivers that carry out the I/O to the device set the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

Otherwise, the miniclass driver should ensure that the **Information** field of the status block is correct in the completed IRP but the miniclass driver should not alter the contents of the **Status** field.

IOCTL_HID_GET_STRING

Operation

Requests that the HID minidriver retrieve a human-readable string for either the manufacturer ID, the product ID, or the serial number from the string descriptor of the device. The minidriver must send a Get String Descriptor request to the device, in order to retrieve the string descriptor, then it must extract the string at the appropriate index from the string descriptor and return it in the output buffer indicated by the IRP. Before sending the Get String Descriptor request, the minidriver must retrieve the appropriate index for the manufacturer ID, the product ID or the serial number from the device extension of a top level collection associated with the device.

Input

IOCTL_HID_GET_STRING makes use of two input buffers.

Parameters.DeviceIoControl.OutputBufferLength in the I/O stack location of the IRP indicates the size in bytes of the locked-down output buffer at **Irp->UserBuffer**.

Parameters.DeviceIoControl.Type3InputBuffer in the I/O stack location of the IRP contains a composite value. The two most significant bytes contain the language ID of the string to be retrieved. The two least significant bytes contain one of the following three constant values:

HID_STRING_ID_IMANUFACTURER
HID_STRING_ID_IPRODUCT
HID_STRING_ID_ISERIALNUMBER

The HID minidriver must determine which of these three constants is present in the lower two bytes of the input buffer, then it must retrieve the corresponding string index from the device descriptor. Device descriptor information is stored in the device extension of a top level collection associated with the device.

It is important not to confuse these three constants with the actual string indices of the IDs. These constants represent the offsets in the device descriptor where the corresponding string indices can be found.

For example, HID_STRING_ID_IMANUFACTURER indicates the location in the device descriptor where the index for the manufacturer ID is found. This index, in turn, serves as an offset into the string descriptor where the human-readable form of the manufacturer ID is located.

Output

The miniclass driver fills the buffer at **Irp->UserBuffer** with the requested string.

I/O Status Block

Miniclass drivers that carry out the I/O to the device set the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

Otherwise, the miniclass driver should ensure that the **Information** field of the status block is correct in the completed IRP but the miniclass driver must not alter the contents of the **Status** field.

IOCTL_HID_READ_REPORT

Operation

Return a report from the device into a class driver-supplied buffer.

Input

Parameters.DeviceIoControl.InputBufferLength contains the size of the buffer provided at **Irp->UserBuffer**.

Output

The miniclass driver fills the system-resident buffer pointed to by **Irp->UserBuffer** with the report data retrieved from the device.

I/O Status Block

Miniclass drivers that carry out the I/O to the device set the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred from the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

Miniclass drivers that call other drivers with this IRP to carry out the I/O to their device should ensure that the **Information** field of the status block is correct and not alter the contents of the **Status** field.

IOCTL_HID_SET_FEATURE

Operation

Sends a feature report packet to a HID device.

Input

Irp->UserBuffer points to a HID_XFER_PACKET structure the contains the parameters and a pointer to a buffer containing the feature report. The following members are used:

- **reportBuffer** points to a buffer containing the report to send to the device.
- **reportBufferLen** contains the length, in bytes, of the buffer provided at **reportBuffer**.
- **reportId** contains the report identifier, for this collection, of the feature report to be retrieved.

Output

None

I/O Status Block

Miniclass drivers that carry out the I/O to the device set the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred to the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

Miniclass drivers that call other drivers with this IRP to carry out the I/O should ensure that the **Information** field of the status block is correct and not alter the contents of the **Status** field.

IOCTL_HID_WRITE_REPORT

Operation

Transmits a class driver-supplied report to the device.

Input

Irp->UserBuffer points to a HID_XFER_PACKET structure the contains the parameters and report to be transmitted to the device. The following members are used:

- **reportBuffer** points to a resident buffer containing the data to be sent to the device.
- **reportBufferLen** contains the length of the buffer provided at **reportBuffer**.
- **reportId** contains the report identifier, for this collection, of the report data to be written to the device.

Output

None

I/O Status Block

Miniclass drivers that carry out the I/O to the device set the following fields of **Irp.IoStatus**:

- **Information** is set to the number of bytes transferred to the device.
- **Status** is set to STATUS_SUCCESS if the transfer completed without error. Otherwise, it is set to an appropriate NTSTATUS error code.

Miniclass drivers that call other drivers with this IRP to carry out the I/O should ensure that the **Information** field of the status block is correct and not alter the contents of the **Status** field.

CHAPTER 2

HID Support Routines for Clients

Human input device (HID) clients can call the following routines to support HID devices.

This chapter covers routines for both user mode and kernel mode clients. As a general rule, user mode clients may call any routine in this chapter. However, kernel mode clients may only call routines that begin with the prefix HidP. See the individual routine comments to determine the availability of a routine.

Routines in this chapter are listed in alphabetical order.

HidD_FlushQueue

```
BOOLEAN
  HidD_FlushQueue(
  IN HANDLE HidDeviceObject
  );
```

HidD_FlushQueue causes a HID drivers to delete all pending information from an input queue of a HID device.

Parameters

HidDeviceObject

Specifies the open handle of a HID device for which the queue is to be flushed.

Return Value

HidD_FlushQueue returns TRUE if the routine completed without error.

Comments

Only user mode clients can call **HidD_FlushQueue**.

HidD_FreePreparsedData

```
BOOLEAN
  HidD_FreePreparsedData(
    IN PHIDP_PREPARSED_DATA PreparsedData
    );
```

HidD_FreePreparsedData releases the resources allocated to hold preparsed data for the HID device.

Parameters

PreparsedData

Points to a buffer, returned from **HidD_GetPreparsedData**, that is to be freed.

Return Value

HidD_FreePreparsedData returns TRUE if the buffer was freed successfully. If FALSE is returned it indicates that the buffer was not a preparsed data buffer.

Comments

Only user mode clients can call **HidD_FreePreparsedData**.

See Also

HidD_GetPreparsedData

HidD_GetAttributes

```
BOOLEAN
  HidD_GetAttributes(
    IN HANDLE HidDeviceObject,
    OUT PHIDD_ATTRIBUTES Attributes
    );
```

HidD_GetAttributes retrieves device attributes for the specified HID device.

Parameters

HidDeviceObject

Specifies the open handle of a HID device for which the attributes are retrieved.

Attributes

Points to a caller-allocated HIDD_ATTRIBUTES structure that is filled with the attribute information for the HID device specified by *HidDeviceObject*.

Return Value

HidD_GetAttributes returns TRUE if the attribute structure provided in *Attributes* was filled without error.

Comments

Only user mode clients can call **HidD_GetAttributes**.

See Also

HIDD_ATTRIBUTES

HidD_GetConfiguration

```
BOOLEAN
  HidD_GetConfiguration(
    IN HANDLE HidDeviceObject,
    IN PHIDD_CONFIGURATION Configuration,
    IN ULONG ConfigurationLength
    );
```

HidD_GetConfiguration is not currently implemented and is reserved for future use.

HidD_GetFeature

```
BOOLEAN
  HidD_GetFeature(
    IN HANDLE HidDeviceObject,
    OUT PVOID ReportBuffer,
    IN ULONG ReportBufferLength
    );
```

HidD_GetFeature retrieves a feature report from a given HID device.

Parameters

HidDeviceObject

Specifies an open handle to a HID device from which to retrieve the feature report.

ReportBuffer

Points to a caller-allocated buffer to hold the feature report from the device.

ReportBufferLength

Specifies the length, in bytes, of the buffer at *ReportBuffer*.

Return Value

HidD_GetFeature returns TRUE if the operation completed without error.

Comments

To retrieve a feature report, the caller must allocate a buffer that is one byte larger than the length of the feature report being retrieved. Before calling this routine, the caller must set the first byte in the buffer to be the report ID of the feature report to be retrieved.

Only user mode clients can call **HidD_GetFeature**.

See Also

HidD_SetFeature

HidD_GetHidGuid

```
VOID
  HidD_GetHidGuid(
    OUT LPGUID HidGuid
    );
```

HidD_GetHidGuid returns the GUID associated with HID devices.

Parameters

HidGuid

Points to a variable to hold the GUID that is associated with all HID devices.

Comments

Only user mode clients can call **HidD_GetHidGuid**.

HidD_GetIndexedString

```
BOOLEAN
  HidD_GetIndexedString(
    IN HANDLE HidDeviceObject,
    IN ULONG StringIndex,
    OUT PVOID Buffer,
    IN ULONG BufferLength
    ) ;
```

HidD_GetIndexedString retrieves an embedded string from a device. The string is selected by a given index value.

Parameters

HidDeviceObject

Specifies an open handle to a HID device from which to retrieve the embedded string.

StringIndex

Specifies the device-specific index of the embedded string to retrieve.

Buffer

Points to a caller-allocated buffer that, on successful return, contains the embedded string retrieved from the device.

BufferLength

Specifies the length, in bytes, of the caller-allocated buffer provided at *Buffer*.

Return Value

HidD_GetIndexedString returns TRUE if the routine completed without error.

Comments

Only user mode clients can call **HidD_GetIndexedString**.

See Also

HidD_GetManufacturerString, **HidD_GetPhysicalDescriptor**, **HidD_GetProductString**, **HidD_GetSerialNumberString**

HidD_GetManufacturerString

```
BOOLEAN
  HidD_GetManufacturerString(
    IN HANDLE HidDeviceObject,
    OUT PVOID Buffer,
    IN ULONG BufferLength
    ) ;
```

HidD_GetManufacturerString retrieves the device-defined embedded string that identifies the manufacturer of a given HID device.

Parameters

HidDeviceObject

Specifies an open handle to a HID device from which to retrieve the manufacturer string.

Buffer

Points to a caller-allocated buffer that, on successful return, contains the embedded string retrieved from the device.

BufferLength

Specifies the length, in bytes, of the caller-allocated buffer provided at *Buffer*.

Return Value

HidD_GetManufacturerString returns TRUE if the routine completed without error.

Comments

Only user mode clients can call **HidD_GetManufacturerString**.

See Also

HidD_GetIndexedString, **HidD_GetPhysicalDescriptor**, **HidD_GetProductString**, **HidD_GetSerialNumberString**

HidD_GetNumInputBuffers

```
BOOLEAN
  HidD_GetNumInputBuffers(
    IN HANDLE HidDeviceObject,
    OUT PULONG NumberBuffers
    ) ;
```

HidD_GetNumInputBuffers retrieves the current size of the ring buffer in the HID class driver which stores packets from a HID device.

Parameters

HidDeviceObject

Specifies an open handle to a HID device from which to retrieve the ring buffer size.

NumberBuffers

Points to a caller-allocated variable that, on return, contains the maximum number of packets the ring buffer holds.

Return Value

HidD_GetNumInputBuffers returns TRUE if the routine completed without error.

Comments

Only user mode clients can call **HidD_GetNumInputBuffers**.

See Also

HidD_SetNumInputBuffers

HidD_GetPhysicalDescriptor

```
BOOLEAN
  HidD_GetPhysicalDescriptor(
    IN HANDLE HidDeviceObject,
    OUT PVOID Buffer,
    IN ULONG BufferLength
    ) ;
```

HidD_GetPhysicalDescriptor retrieves a device-defined embedded string that identifies the physical device. The contents of this string are device-specific.

Parameters

HidDeviceObject

Specifies an open handle to a HID device from which to retrieve the physical descriptor.

Buffer

Points to a caller allocated buffer that, on successful completion, contains the requested descriptor.

BufferLength

Specifies the length, in bytes, of the buffer at *Buffer*.

Return Value

HidD_GetPhysicalDescriptor returns TRUE if the routine completed without error.

Comments

Only user mode clients can call **HidD_GetPhysicalDescriptor**.

See Also

HidD_GetIndexedString, **HidD_GetManufacturerString**, **HidD_GetProductString**, **HidD_GetSerialNumberString**

HidD_GetPreparsedData

```
BOOLEAN
  HidD_GetPreparsedData(
    IN HANDLE HidDeviceObject,
    OUT PHIDP_PREPARSED_DATA *PreparsedData
    );
```

HidD_GetPreparsedData retrieves preparsed data that is used to describe the data returned from a HID device.

Parameters

HidDeviceObject

Specifies the open handle of a HID device from which the preparsed data is to be retrieved.

PreparsedData

Points to a variable to hold the address of a routine-allocated buffer containing the preparsed data from the device.

Return Value

HidD_GetPreparsedData returns TRUE if the routine completed without error.

Comments

Only user mode clients can call **HidD_GetPreparsedData**.

When the preparsed data returned by this routine is no longer needed, clients should call **HidD_FreePreparsedData** to free the buffer allocated for the preparsed data.

See Also

HidD_FreePreparsedData

HidD_GetProductString

```
BOOLEAN
  HidD_GetProductString(
    IN HANDLE HidDeviceObject,
    OUT PVOID Buffer,
    IN ULONG BufferLength
    ) ;
```

HidD_GetProductString retrieves a device-defined embedded string that identifies the manufacturer's product. The contents of this string are manufacturer-determined.

Parameters

HidDeviceObject

Specifies the open handle of a HID device from which the embedded product string is to be retrieved.

Buffer

Points to a caller allocated buffer that, on successful completion, contains the requested string.

BufferLength

Specifies the length, in bytes, of the buffer at *Buffer*.

Return Value

HidD_GetProductDescriptor returns TRUE if the routine completed without error.

Comments

Only user mode clients can call **HidD_GetProductString**.

See Also

HidD_GetIndexedString, **HidD_GetManufacturerString**, **HidD_GetPhysical-Descriptor**, **HidD_GetSerialNumberString**

HidD_GetSerialNumberString

```
BOOLEAN
  HidD_GetSerialNumberString(
    IN HANDLE HidDeviceObject,
    OUT PVOID Buffer,
    IN ULONG BufferLength
    ) ;
```

HidD_GetSerialNumberString retrieves a device-defined embedded string that contains the serial number of the device. The contents of this string are manufacturer-determined.

Parameters

HidDeviceObject

Specifies the open handle to a HID device from which the embedded serial number string is to be retrieved.

Buffer

Points to a caller allocated buffer that, on successful completion, contains the requested string.

BufferLength

Specifies the length, in bytes, of the buffer at *Buffer*.

Return Value

HidD_GetSerialNumberString returns TRUE if the routine completed without error.

Comments

Only user mode clients can call **HidD_GetSerialNumberString**.

See Also

HidD_GetIndexedString, **HidD_GetManufacturerString**, **HidD_GetPhysical-Descriptor**, **HidD_GetProductString**

HidD_SetConfiguration

```
BOOLEAN
  HidD_SetConfiguration(
    IN HANDLE HidDeviceObject,
    IN PHIDD_CONFIGURATION Configuration,
    IN ULONG ConfigurationLength
    );
```

HidD_SetConfiguration is not currently implemented and is reserved for future use.

HidD_SetConfiguration sets a specified configuration on a HID device.

Parameters

HidDeviceObject

Specifies the open handle of a HID device for which the configuration is to be set.

Configuration

Points to a caller-allocated buffer that contains the new configuration for the HID device.

ConfigurationLength

Specifies the length, in bytes, of the buffer specified at *Configuration*.

Return Value

HidD_SetConfiguration returns TRUE if the routine completed without error. If FALSE is returned, either **HidD_GetConfiguration** was not called before this routine or the device returned an error.

Comments

Only user mode clients can call **HidD_SetConfiguration**.

The buffer specified at *Configuration* must contain the configuration information retrieved from **HidD_GetConfiguration** after being adjusted for the new settings. Failure to call **HidD_GetConfiguration** first and using the data returned will cause **HidD_SetConfiguration** to fail.

See Also

HidD_GetConfiguration

HidD_SetFeature

```
BOOLEAN
  HidD_SetFeature(
    IN HANDLE HidDeviceObject,
    IN PVOID ReportBuffer,
    IN ULONG ReportBufferLength
    );
```

HidD_SetFeature sends a feature report to a given HID device.

Parameters

HidDeviceObject

Specifies an open handle to a HID device from which to retrieve the feature report.

ReportBuffer

Points to a caller-allocated buffer to hold the feature report for the device.

ReportBufferLength

Specifies the length, in bytes, of the buffer at *ReportBuffer*.

Return Value

HidD_SetFeature returns TRUE if the operation completed without error.

Comments

Before calling this routine, the caller can set the first byte of the buffer to prepare a specific report ID of the feature report being sent.

Only user mode clients can call **HidD_SetFeature**.

See Also

HidD_GetFeature

HidD_SetNumInputBuffers

```
BOOLEAN
  HidD_SetNumInputBuffers(
    IN HANDLE HidDeviceObject,
    OUT ULONG NumberBuffers
    ) ;
```

HidD_SetNumInputBuffers sets the maximum number of packets that the HID class driver ring buffer holds for the given device.

Parameters

HidDeviceObject

Specifies an open handle to a HID device on which to set the maximum ring buffer size.

NumberBuffers

Specifies the maximum number of buffers that the HID class driver should maintain for data from this device.

Return Value

HidD_SetNumInputBuffers returns TRUE if the operation completed without error.

Comments

If **HidD_SetNumInputBuffers** returns FALSE and **GetLastError** indicates that an invalid parameter was supplied, the number of input buffers specified was below the minimum number allowed by the HID class driver.

See Also

HidD_GetNumInputBuffers

HidP_GetButtonCaps

```
NTSTATUS
  HidP_GetButtonCaps(
    IN HIDP_REPORT_TYPE ReportType,
    OUT PHIDP_BUTTON_CAPS ButtonCaps,
    IN OUT PULONG ButtonCapsLength,
    IN PHIDP_PREPARSED_DATA PreparsedData
    );
```

HidP_GetButtonCaps returns the capabilities for all buttons for a given top level collection.

Parameters

ReportType

Specifies the type of report for which to retrieve the button capabilities. This parameter must be one of the following:

HidP_Input

Specifies that **HidP_GetButtonCaps** return the button capabilities for all input reports.

HidP_Output

Specifies that **HidP_GetButtonCaps** return the button capabilities for all output reports.

HidP_Feature

Specifies that **HidP_GetButtonCaps** return the button capabilities for all feature reports.

ButtonCaps

Points to a caller-allocated buffer that will contain, on return, an array of HIDP_BUTTON_CAPS structures that contain information for all buttons on the HID device.

ButtonCapsLength

Specifies the length on input, *in array elements*, of the buffer provided at *ButtonCaps*. On output, this parameter is set to the actual number of elements that were returned by this routine, in the buffer provided at *ButtonCaps* if the routine completed without error.

The correct length necessary to retrieve the button capabilities can be found in the capability data returned for the device by **HidP_GetCaps**.

PreparsedData

Points to the preparsed data returned for the device when top-level collection information was obtained at initialization.

Return Value

HidP_GetButtonCaps returns one of the following HIDP_XXX status codes:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

Comments

This routine will retrieve the capability data for all buttons in the top level collection without regard to the usage, usage page, or link collection. To retrieve button capabilities for a specific usage, usage page, or link collection use **HidP_GetSpecificButtonCaps** instead.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

HidP_GetSpecificButtonCaps, **HidP_GetCaps**, HIDP_BUTTON_CAPS

HidP_GetButtons

```
NTSTATUS
  HidP_GetButtons(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage,
    IN USHORT LinkCollection    OPTIONAL,
    OUT USAGE *UsageList,
    IN OUT ULONG *UsageLength,
    IN PHIDP_PREPARSED_DATA PreparsedData,
    IN PCHAR Report,
    IN ULONG ReportLength
    );
```

HidP_GetButtons takes a report from a HID device and returns the current state of the buttons in that report.

Parameters

ReportType

Specifies the type of report, provided at *Report*, from which to retrieve the buttons. This parameter must be one of the following values:

HidP_Input

Specifies that the device data report provided at *Report* is an input report.

HidP_Output

Specifies that the device data report provided at *Report* is an output report.

HidP_Feature

Specifies that the device data report provided at *Report* is a feature report.

UsagePage

Specifies the usage page of the buttons for which to retrieve the current state.

LinkCollection

Optionally specifies a link collection identifier used to retrieve only specific button states. If this value is nonzero, only buttons that are part of the given link collection will be returned.

UsageList

Points to a caller-allocated buffer that contains, on return, the usages of all buttons that are pressed which belong to the usage page specified in *UsagePage*.

UsageLength

Is the length, *in array elements*, of the buffer provided at *UsageList*. On return this parameter is set to the number of button states that were set by this routine into the buffer provided at *UsageList*. If the error HIDP_STATUS_BUFFER_TOO_SMALL was returned, this parameter will contain the number of array elements required to hold all button data requested.

The maximum number of buttons that can ever be returned for a given type of report can be obtained by calling **HidP_MaxUsageListLength**.

PreparsedData

Points to the preparsed data returned for the device when collection information was obtained at initialization.

Report

Points to the device data that contains the button states to be retrieved.

ReportLength

Is the length, in bytes, of the buffer provided at *Report*.

Return Value

HidP_GetButtons returns a HIDP_XXX status code from the following list:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_INVALID_REPORT_TYPE

Indicates that the value specified in *ReportType* was invalid.

HIDP_STATUS_BUFFER_TOO_SMALL

Indicates that the buffer provided at *UsageList* is too small to hold the data retrieved. The correct length to retrieve all usages can be found in the capability data returned by **HidP_GetCaps**.

HIDP_INVALID_REPORT_LENGTH

Indicates that the report length provided in *ReportLength* is not the expected length of a report of the type specified in *ReportType*.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

HIDP_STATUS_INCOMPATIBLE_REPORT_ID

Indicates that the buttons states specified by the parameter *UsagePage* is known, but cannot be found in the data provided at *Report*.

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that button states specified by the parameter *UsagePage* cannot be found in any data report for the HID device.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_GetCaps, **HidP_MaxUsageListLength**

HidP_GetButtonsEx

```
NTSTATUS
  HidP_GetButtonsEx(
    IN HIDP_REPORT_TYPE ReportType,
    IN USHORT LinkCollection,
    OUT PUSAGE_AND_PAGE ButtonList,
    IN OUT ULONG *UsageLength,
    IN PHIDP_PREPARSED_DATA PreparsedData,
    IN PCHAR Report,
    IN ULONG ReportLength
    );
```

HidP_GetButtonsEx takes a report from a HID device and returns the current state of all the buttons in the given data report.

Parameters

ReportType

Specifies the type of report, provided at *Report*, from which to retrieve the button states. This parameter must be one of the following values:

HidP_Input

Specifies that the device data report provided at *Report* is an input report.

HidP_Output

Specifies that the device data report provided at *Report* is an output report.

HidP_Feature

Specifies that the device data report provided at *Report* is a feature report.

LinkCollection

Optionally specifies a link collection identifier used to retrieve only specific button states. If this value is nonzero, only buttons that are part of the given link collection will be returned.

ButtonList

Points to a caller-allocated buffer that contains, on return, the usage and usage page values of each button that is down.

UsageLength

Is the length, *in array elements*, of the buffer provided at *UsageList*. On return this parameter is set to the number of button states that were set by this routine into the buffer provided at *UsageList*. If the error HIDP_STATUS_BUFFER_TOO_SMALL was returned, this parameter will contain the number of array elements required to hold all button data requested.

The maximum number of buttons that can ever be returned for a given type of report can be obtained by calling **HidP_MaxUsageListLength** or by passing zero as the **UsagePage** member in *ButtonList*.

PreparsedData

Points to the preparsed data returned for the device when collection information was obtained at initialization.

Report

Points to the device data that contains the button states to be retrieved.

ReportLength

Is the length, in bytes, of the buffer provided at *Report*.

Return Value

HidP_GetButtonsEx returns a HIDP_XXX status code from the following list:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_INVALID_REPORT_TYPE

Indicates that the value specified in *ReportType* was invalid.

HIDP_STATUS_BUFFER_TOO_SMALL

Indicates that the buffer provided at *UsageList* is too small to hold the data retrieved. The correct length to retrieve all usages can be found in the capability data returned by **HidP_GetCaps**.

HIDP_INVALID_REPORT_LENGTH

Indicates that the report length provided in *ReportLength* is not the expected length of a report of the type specified in *ReportType*.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that no buttons could be found for this HID device.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_GetCaps, **HidP_MaxUsageListLength**, USAGE_AND_PAGE

HidP_GetCaps

```
NTSTATUS
  HidP_GetCaps(
    IN PHIDP_PREPARSED_DATA PreparsedData,
    OUT PHIDP_CAPS Capabilities
    );
```

HidP_GetCaps returns the capabilities of a HID device based on the given preparsed data.

Parameters

PreparsedData

Points to a caller-allocated buffer that holds the top level collection description retrieved from the HID device.

Capabilities

Points to a caller-allocated buffer that, on return, contains the parsed capability information for this HID device.

Return Value

HidP_GetCaps returns HIDP_STATUS_SUCCESS if the routine completed parsing of the data successfully. HIDP_STATUS_INVALID_PREPARSED_DATA is returned if the preparsed data pointed to by *PreparsedData* is malformed.

Comments

Drivers obtain the top level collection data used at *PreparsedData* by calling **HidD_GetPreparsedData**.

For kernel mode clients, the caller-allocated buffer supplied at *PreparsedData* must be allocated from nonpaged pool.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

HidD_GetPreparsedData, HIDP_CAPS

HidP_GetLinkCollectionNodes

```
NTSTATUS
  HidP_GetLinkCollectionNodes(
    OUT PHIDP_LINK_COLLECTION_NODE LinkCollectionNodes,
    IN OUT PULONG LinkCollectionNodesLength,
    IN PHIDP_PREPARSED_DATA PreparsedData
    );
```

HidP_GetLinkCollectionNodes returns an array of LINK_COLLECTION_NODE structures that describes the relationships and layout of the link collections within this top level collection.

Parameters

LinkCollectionNodes

Points to a caller-allocated array of HIDP_LINK_COLLECTION_NODE structures in which **HidP_GetLinkCollectionNodes** returns an entry for each collection within the top-level collection.

LinkCollectionNodesLength

Specifies, on input, the length, *in array elements*, of the buffer provided at *LinkCollectionNodes*. On output, this parameter is set to the number of entries in the array at *LinkCollectionNodes* that were initialized.

PreparsedData

Points to the preparsed data returned for the device when top-level collection information was obtained at initialization.

Return Value

HidP_GetLinkCollectionNodes returns one of the following HIDP_XXX status codes:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_BUFFER_TOO_SMALL

Indicates that the buffer provided at *LinkCollectionNodes* is too small to hold all entries for the link collection nodes. *LinkCollectionNodesLength* is set to the length, *in array elements*, required to hold the link collection nodes information.

Comments

The length of the buffer required, *in array elements*, for an entire link collection node array is found in the HIDP_CAPS structure member **NumberLinkCollectionNodes**. Clients obtain HIDP_CAPS information by calling **HidP_GetCaps**.

For information on the relationships of link collections described by the data returned from this routine, see HIDP_LINK_COLLECTION_NODE.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_GetCaps, HIDP_CAPS, HIDP_LINK_COLLECTION_NODE

HidP_GetScaledUsageValue

```
NTSTATUS
  HidP_GetScaledUsageValue(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage,
    IN USHORT LinkCollection    OPTIONAL,
    IN USAGE Usage,
    OUT PLONG UsageValue,
    IN PHIDP_PREPARSED_DATA PreparsedData,
    IN PCHAR Report,
    IN ULONG ReportLength
    );
```

HidP_GetScaledUsageValue returns the signed result of a value, adjusted for the scaling factor, retrieved from a report packet from the device.

Parameters

ReportType

Specifies the type of report, provided at *Report*, from which to retrieve the scaled value. This parameter must be one of the following values:

HidP_Input

Specifies that the report provided at *Report* is an input report.

HidP_Output

Specifies that the report provided at *Report* is an output report.

HidP_Feature

Specifies that the report provided at *Report* is an feature report.

UsagePage

Specifies the usage page of the value to be retrieved.

LinkCollection

Optionally specifies the link collection identifier of the value to be retrieved.

Usage

Specifies the usage of the scaled valued to retrieve.

UsageValue

Points to a variable, that on return from this routine, holds the scaled value retrieved from the device report.

PreparsedData

Points to the preparsed data for the device.

Report

Points to a caller-allocated buffer that contains device report data.

ReportLength

Specifies the length, in bytes, of the report data provided at *Report*.

Return Value

HidP_GetScaledUsageValue returns one of the following HIDP_XXX status codes:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_REPORT_TYPE

Indicates that the value specified in *ReportType* is invalid.

HIDP_STATUS_INVALID_REPORT_LENGTH

Indicates that the report length provided in *ReportLength* is not the expected length of a report of the type specified in *ReportType*.

HIDP_STATUS_BAD_LOG_PHY_VALUES

Indicates that the device has returned an illegal logical and physical value preventing scaling. To retrieve the values returned by the device call **HidP_GetUsageValue** instead.

HIDP_STATUS_NULL

Indicates the current state of the scaled value data from the device is less than the logical minimum or is greater than the logical maximum but the scaled value has a NULL state.

HIDP_STATUS_VALUE_OUT_OF_RANGE

Indicates the current state of the scaled value data from the device is less than the logical minimum or is greater than the logical maximum.

HIDP_STATUS_INCOMPATIBLE_REPORT_ID

Indicates that the value specified by the parameters *Usage, UsagePage,* and optionally *LinkCollection* is known, but cannot be found in the data provided at *Report*.

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that the data specified by the values provided at *Usage, UsagePage*, and optionally *LinkCollection* cannot be found in any data report.

Comments

The caller-allocated buffers supplied at *PreparsedData, UsageValue,* and *Report* must be allocated from non-paged pool.

Callers who wish to obtain all data for a usage that contains multiple data items for a single usage, that corresponds to a HID byte array, must call **HidP_GetUsageValueArray** instead.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_GetUsageValue

HidP_GetSpecificButtonCaps

```
NTSTATUS
  HidP_GetSpecificButtonCaps(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage,
    IN USHORT LinkCollection,
    IN USAGE Usage,
    OUT PHIDP_BUTTON_CAPS ButtonCaps,
    IN OUT PULONG ButtonCapsLength,
    IN PHIDP_PREPARSED_DATA PreparsedData
    );
```

HidP_GetSpecificButtonCaps retrieves the capabilities for all buttons in a specific type of report that meet the search criteria.

Parameters

ReportType

Specifies the type of report for which to retrieve the button capabilities. This parameter must be one of the following values:

HidP_Input

Specifies that **HidP_GetSpecificButtonCaps** return the button capabilities for all input reports.

HidP_Output

Specifies that **HidP_GetSpecificButtonCaps** return the button capabilities for all output reports.

HidP_Feature

Specifies that **HidP_GetSpecificButtonCaps** return the button capabilities for all feature reports.

UsagePage

Specifies a usage page identifier to use as a search criteria. If this parameter is nonzero, then only buttons that specify this usage page will be retrieved.

LinkCollection

Specifies a link collection identifier to use as a search criteria. If this parameter is nonzero, then only buttons that are part of this link collection will be retrieved.

Usage

Specifies a usage identifier to use as a search criteria. If this parameter is nonzero, then only buttons that specify this usage will be retrieved.

ButtonCaps

Points to a caller-allocated buffer that will contain, on return, an array of HIDP_BUTTON_CAPS structures that contain information for all buttons that meet the search criteria.

ButtonCapsLength

Specifies the length on input, *in array elements*, of the buffer provided at *ButtonCaps*. On output, this parameter is set to the actual number of elements that were set by this routine, into the buffer provided at *ButtonCaps*.

The correct length necessary to retrieve the button capabilities can be found in the capability data returned for the device by **HidP_GetCaps**.

PreparsedData

Points to the preparsed data returned for the device when top level collection information was obtained at initialization.

Return Value

HidP_GetSpecificButtonCaps returns one of the following HIDP_XXX status codes:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

Comments

HidP_GetSpecificButtonCaps is used as a search routine to retrieve button capability data for buttons that meet given search criteria as opposed to **HidP_GetButtonCaps** which returns the capability data for all buttons on the device. Calling **HidP_GetSpecificButtonCaps** specifying zero for *UsagePage, Usage,* and *LinkCollection* is functionally equivalent to calling **HidP_GetButtonCaps**.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

HidP_GetButtonCaps, **HidP_GetCaps**, HIDP_BUTTON_CAPS

HidP_GetSpecificValueCaps

```
NTSTATUS
  HidP_GetSpecificValueCaps(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage,
    IN USHORT LinkCollection,
    IN USAGE Usage,
    OUT PHIDP_VALUE_CAPS ValueCaps,
    IN OUT PULONG ValueCapsLength,
    IN PHIDP_PREPARSED_DATA PreparsedData
    );
```

HidP_GetSpecificValueCaps retrieves the capabilities for all values in a specific type of report that meet the search criteria.

Parameters

ReportType

Specifies the type of report for which to retrieve the value capabilities. This parameter must be one of the following values:

HidP_Input

Specifies that **HidP_GetSpecificValueCaps** return the value capabilities for all input reports.

HidP_Output

Specifies that **HidP_GetSpecificValueCaps** return the value capabilities for all output reports.

HidP_Feature

Specifies that **HidP_GetSpecificValueCaps** return the value capabilities for all feature reports.

UsagePage

Specifies a usage page identifier to use as a search criteria. If this parameter is nonzero, then only values that specify this usage page will be retrieved.

LinkCollection

Specifies a link collection identifier to use as a search criteria. If this parameter is nonzero, then only values that are part of this link collection will be retrieved.

Usage

Specifies a usage identifier to use as a search criteria. If this parameter is nonzero, then only buttons that specify this value will be retrieved.

ValueCaps

Points to a caller-allocated buffer that will contain, on return, an array of HIDP_VALUE_CAPS structures that contain information for all values that meet the search criteria.

ValueCapsLength

Specifies the length on input, *in array elements*, of the buffer provided at *ValueCaps*. On output, this parameter is set to the actual number of elements that were set by this routine, into the buffer provided at *ValueCaps*.

The correct length necessary to retrieve the value capabilities can be found in the capability data returned for the device by **HidP_GetCaps**.

PreparsedData

Points to the preparsed data returned for the device when top level collection information was obtained at initialization.

Return Value

HidP_GetSpecificValueCaps returns a HIDP_XXX status code from the following list:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

Comments

HidP_GetSpecificValueCaps is used as a search routine to retrieve button capability data for buttons that meet given search criteria as opposed to **HidP_GetValueCaps** which returns the capability data for all buttons on the device. Calling **HidP_GetSpecificValueCaps**

specifying zero for *UsagePage, Usage,* and *LinkCollection* is functionally equivalent to calling **HidP_GetValueCaps**.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

HidP_GetCaps, **HidP_GetValueCaps**, HIDP_VALUE_CAPS

HidP_GetUsageValue

```
NTSTATUS
  HidP_GetUsageValue(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage,
    IN USHORT LinkCollection,
    IN USAGE Usage,
    OUT PULONG UsageValue,
    IN PHIDP_PREPARSED_DATA PreparsedData,
    IN PCHAR Report,
    IN ULONG ReportLength
    );
```

HidP_GetUsageValue returns a value from a device data report given a selected search criteria.

Parameters

ReportType

Specifies the type of report, provided at *Report*, from which to retrieve the value. This parameter must be one of the following values:

HidP_Input

Specifies that the device data report provided at *Report* is an input report.

HidP_Output

Specifies that the device data report provided at *Report* is an output report.

HidP_Feature

Specifies that the device data report provided at *Report* is a feature report.

UsagePage

Specifies the usage page identifier of the value to retrieve.

LinkCollection

Optionally specifies the link collection identifier of the value to be retrieved.

Usage

Specifies the usage of the scaled valued to retrieve.

UsageValue

Points to a variable, that on return from this routine, holds the value retrieved from the device report.

PreparsedData

Points to the preparsed data returned for the device when collection information was obtained at initialization.

Report

Points to the device data that contains the value to be retrieved.

ReportLength

Is the length, in bytes, of the buffer provided at *Report*.

Return Value

HidP_GetUsageValue returns a HIDP_XXX status code from the following list:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

HIDP_STATUS_INVALID_REPORT_TYPE

Indicates that the value specified in *ReportType* was invalid.

HIDP_STATUS_INVALID_REPORT_LENGTH

Indicates that the report length provided in *ReportLength* is not the expected length of a report of the type specified in *ReportType*.

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that the usage specified by the values provided at *Usage, UsagePage*, and optionally, *LinkCollection* could not be found in the report data provided.

HIDP_STATUS_INCOMPATIBLE_REPORT_ID

Indicates that the value specified by the parameters *Usage, UsagePage,* and optionally *LinkCollection* is known, but cannot be found in the data provided at *Report*.

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that the data specified by the values provided at *Usage, UsagePage*, and optionally *LinkCollection* cannot be found in any data report.

Comments

This routine does not sign the value. To have the sign bit automatically applied, use the routine **HidP_GetScaledUsageValue** instead. For manually assigning the sign bit, the position of the sign bit can be found in the HIDP_VALUE_CAPS structure for this value.

Callers who wish to obtain all data for a usage that contains multiple data items for a single usage, that corresponds to a HID byte array, must call **HidP_GetUsageValueArray** instead.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_GetScaledUsageValue, HIDP_VALUE_CAPS

HidP_GetUsageValueArray

```
NTSTATUS
  HidP_GetUsageValueArray(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage,
    IN USHORT LinkCollection    OPTIONAL,
    IN USAGE Usage,
    OUT PCHAR UsageValue,
    IN USHORT UsageValueByteLength,
    IN PDIP_PREPARSED_DATA PreparsedData,
    IN PCHAR Report,
    IN ULONG ReportLength
    );
```

HidP_GetUsageValueArray returns the device data for a usage that contains multiple data items for a single usage.

Parameters

ReportType

Specifies the type of report, provided at *Report*, from which to retrieve the values. This parameter must be one of the following values:

HidP_Input

Specifies that the device data report provided at *Report* is an input report.

HidP_Output

Specifies that the device data report provided at *Report* is an output report.

HidP_Feature

Specifies that the device data report provided at *Report* is a feature report.

UsagePage

Specifies the usage page identifier of the data to be retrieved.

LinkCollection

Optionally specifies the link collection identifier of the data to be retrieved.

Usage

Specifies the usage identifier of the valued to retrieve.

UsageValue

Points to a caller-allocated buffer that contains, on output, the data from the device. The correct length for this buffer can be found by multiplying the **ReportCount** and **BitSize** fields of the HIDP_VALUE_CAPS structure for this value and rounding the resultant value up to the nearest byte.

UsageValueByteLength

Specifies the length, in bytes, of the buffer at *UsageValue*.

PreparsedData

Points to the preparsed data returned for the device when collection information was obtained at initialization.

Report

Points to the device data that contains the data to be retrieved.

ReportLength

Is the length, in bytes, of the buffer provided at *Report*.

ReturnValue

HidP_GetUsageValueArray returns a HIDP_XXX status code from the following list:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

HIDP_STATUS_INVALID_REPORT_TYPE

Indicates that the value specified in *ReportType* was invalid.

HIDP_STATUS_INVALID_REPORT_LENGTH

Indicates that the report length provided in *ReportLength* is not the expected length of a report of the type specified in *ReportType*.

HIDP_STATUS_NOT_VALUE_ARRAY

Indicates that the requested usage has only one data item. To retrieve the data, clients should call **HidP_GetUsageValue** or **HidP_GetScaledUsageValue** instead.

HIDP_STATUS_BUFFER_TOO_SMALL

Indicates that the buffer provided at *UsageValue* is too small to hold the data requested. The correct length to retrieve all data for this usage can be found by multiplying the values in **BitSize** and **ReportCount,** from the HIDP_VALUE_CAPS structure for this value, together and round the resultant value up to the nearest byte.

HIDP_STATUS_INCOMPATIBLE_REPORT_ID

Indicates that the value specified by the parameters *Usage, UsagePage,* and optionally *Link-Collection* is known, but cannot be found in the data provided at *Report.*

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that the data specified by the values provided at *Usage, UsagePage*, and optionally *LinkCollection* cannot be found in any data report for this top-level collection.

Comments

When **HidP_GetUsageValueArray** retrieves the data, it will fill in the buffer in little-endian order beginning with the least significant bit of the data for this usage. The data is filled in without regard to byte alignment and is shifted such that the least significant bit is placed as the 1st bit of the given buffer.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_GetScaledUsageValue, **HidP_GetUsageValue**

HidP_GetValueCaps

```
NTSTATUS
  HidP_GetValueCaps(
    IN HIDP_REPORT_TYPE ReportType,
    OUT PHIDP_VALUE_CAPS ValueCaps,
    IN OUT PULONG ValueCapsLength,
    IN PHIDP_PREPARSED_DATA PreparsedData
    );
```

HidP_GetValueCaps retrieves the capabilities of all values for a given top level collection.

Parameters

ReportType

Specifies the type of report for which to retrieve the value capabilities. This parameter must be one of the following values:

HidP_Input

Specifies that **HidP_GetUsages** return the value capabilities for all input reports.

HidP_Output

Specifies that **HidP_GetUsages** return the value capabilities for all output reports.

HidP_Feature

Specifies that **HidP_GetUsages** return the value capabilities for all feature reports.

ValueCaps

Points to a caller-allocated buffer that will contain, on return, an array of HIDP_VALUE_CAPS structures that contain information for all values in the top level collection.

ValueCapsLength

Specifies the length on input, *in array elements*, of the buffer provided at *ValueCaps*. On output, this parameter is set to the actual number of elements that were set by this routine, into the buffer provided at *ValueCaps*.

The correct length necessary to retrieve the value capabilities can be found in the capability data returned for the device by **HidP_GetCaps**.

PreparsedData

Points to the preparsed data returned for the device when collection information was obtained at initialization.

Return Value

HidP_GetValueCaps returns a HIDP_XXX status code from the following list:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

Comments

This routine will retrieve the capability data for all values in the top level collection without regard to the usage, usage page, or link collection of the value. To retrieve value capabilities for a specific usage, usage page, or link collection use **HidP_GetSpecificValueCaps** instead.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_GetCaps, **HidP_GetSpecificValueCaps**, HIDP_VALUE_CAPS

HidP_MaxUsageListLength

```
ULONG
  HidP_MaxUsageListLength(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage    OPTIONAL,
    IN PHIDP_PREPARSED_DATA PreparsedData
    );
```

HidP_MaxUsageListLength returns the maximum number of buttons that can be returned from a given report type for the top level collection.

Parameters

ReportType

Specifies the report type for which to get a maximum usage count. *ReportType* must be one of the following values:

HidP_Input

Specifies that **HidP_MaxUsageListLength** should return the maximum number of buttons for an input report.

HidP_Output

Specifies that **HidP_MaxUsageListLength** should return the maximum number of buttons for an output report.

HidP_Feature

Specifies that **HidP_MaxUsageListLength** should return the maximum number of buttons for an feature report.

UsagePage

Optionally specifies a usage page identifier to use as a search criteria. If this parameter is zero, the routine returns the number of buttons for the entire top-level collection regardless of usage page.

PreparsedData

Points to the preparsed data returned for the device when collection information was obtained at initialization.

Return Value

HidP_MaxUsageLength returns the maximum number of buttons, that are of the given usage page, that will be returned in a given report type. If the buffer provided at *PreparsedData* or the value of *ReportType* is invalid, zero is returned.

Comments

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

HidP_SetButtons

```
NTSTATUS
  HidP_SetButtons(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage,
    IN USHORT LinkCollection    OPTIONAL,
    IN PUSAGE UsageList,
    IN OUT PULONG UsageLength,
    IN PHIDP_PREPARSED_DATA PreparsedData,
    IN OUT PCHAR Report,
    IN ULONG ReportLength
    );
```

HidP_SetButtons sets takes an array of button state data and sets the button data in a given report.

Parameters

ReportType

Specifies the type of report provided at *Report*. *ReportType* must be one of the following values:

HidP_Input

Specifies that the report is an input report.

HidP_Output

Specifies that the report is an output report.

HidP_Feature

Specifies that the report is a feature report.

UsagePage

Specifies the usage page identifier of the buttons to be set in the report.

LinkCollection

Optionally specifies a link collection identifier to distinguish between buttons. If this parameter is zero, *LinkCollection* is ignored.

UsageList

Points to a caller-allocated buffer that contains an array of button data to be set in the report provided at *Report*.

UsageLength

Specifies the length, *in array elements*, of the buffer provided at *UsageList*. If an error is returned by this routine, this parameter contains the position in the array provided at *UsageList* where the error was encountered. All previous array entries were successfully set in the report provided at *Report*.

PreparsedData

Points to the preparsed data returned for the device when collection information was obtained at initialization.

Report

Points to a caller-allocated buffer for a report that contains the buttons to be set.

ReportLength

Specifies the length, in bytes, of the buffer provided at *Report*.

Return Value

HidP_SetButtons returns a HIDP_XXX status code from the following list:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_REPORT_TYPE

Indicates that the report type value provided at *ReportType* was invalid.

HIDP_STATUS_INVALID_REPORT_LENGTH

Indicates that the report length provided at *ReportLength* does not match the expected report length for a report of type specified in *ReportType*.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

HIDP_STATUS_BUFFER_TOO_SMALL

Indicates that the buffer provided at *Report* was of insufficient size to store all of the buttons being set. This error is also returned when the report does not have enough locations to set all of the buttons. It is necessary to split this request into two reports.

HIDP_STATUS_INCOMPATIBLE_REPORT_ID

Indicates that the button, at the array element specified on return in *UsageLength*, is a valid button but could not be set in the report provided at *Report* because of previous buttons already set in that report. A new report would need to be allocated for this button.

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that the button, at the array element specified on return in *UsageLength*, is not a valid button for this device and could not be set.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_GetButtons

HidP_SetScaledUsageValue

```
NTSTATUS
  HidP_SetScaledUsageValue(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage,
    IN USHORT LinkCollection    OPTIONAL,
    IN USAGE Usage,
    IN LONG UsageValue,
    IN PHIDP_PREPARSED_DATA PreparsedData,
    IN OUT PCHAR Report,
    IN ULONG ReportLength
    );
```

HidP_SetScaledUsageValue takes a signed physical (scaled) number and converts it to the logical, or device, representation and inserts it in a given report.

Parameters

ReportType

Specifies the type of report provided at *Report*. *ReportType* must be one of the following values:

HidP_Input

Specifies that the report is an input report.

HidP_Output

Specifies that the report is an output report.

HidP_Feature

Specifies that the report is a feature report.

UsagePage

Specifies the usage page identifier of the value to be set in the report.

LinkCollection

Optionally specifies a link collection identifier to distinguish between values that have the same usage page and usage identifiers. If this parameter is zero, *LinkCollection* will be ignored.

Usage

Specifies the usage identifier of the value to be set in the report.

UsageValue

Specifies the physical, or scaled, value to be set in the value for the given report.

PreparsedData

Points to the preparsed data returned for the device when collection information was obtained at initialization.

Report

Points to a caller-allocated buffer for a report that contains the scaled value to be set.

ReportLength

Is the size, in bytes, of the buffer provided at *Report*.

Return Value

HidP_SetScaledUsageValue returns a HIDP_XXX status code from the following list:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

HIDP_STATUS_INVALID_REPORT_LENGTH

Indicates that the report length provided in *ReportLength* was not the size expected for a report of the report type specified in *ReportType*.

HIDP_STATUS_BAD_LOG_PHY_VALUES

Indicates that the device has an illegal logical or physical value preventing scaling. To set the value call **HidP_SetUsageValue** instead.

HIDP_STATUS_NULL

Indicates the given state of the scaled value data from the device is less than the physical minimum or is greater than the physical maximum and the scaled value has a NULL state.

HIDP_STATUS_VALUE_OUT_OF_RANGE

Indicates the current state of the scaled value data from the device is less than the physical minimum or is greater than the physical maximum.

HIDP_STATUS_INCOMPATIBLE_REPORT_ID

Indicates that the value specified by the parameters *Usage, UsagePage,* and optionally *Link-Collection* is known, but cannot be set in the data provided at *Report* because it conflicts with data already in that report. A new report would need to be allocated for this data.

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that the usage specifies by the values provided at *Usage, UsagePage*, and optionally *LinkCollection* could not be found in the report data provided.

Comments

This routine automatically handles the setting of the signed bit in the data to be sent to the device.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_SetUsageValue

HidP_SetUsageValue

```
NTSTATUS
  HidP_SetUsageValue(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage,
    IN USHORT LinkCollection,
    IN USAGE Usage,
    IN ULONG UsageValue,
    IN PHIDP_PREPARSED_DATA PreparsedData,
    IN OUT PCHAR Report,
    IN ULONG ReportLength
    );
```

HidP_SetUsageValue sets a value in a given report.

Parameters

ReportType

Specifies the type of report provided at *Report*. *ReportType* must be one of the following values:

HidP_Input

Specifies that the report is an input report.

HidP_Output

Specifies that the report is an output report.

HidP_Feature

Specifies that the report is a feature report.

UsagePage

Specifies the usage page identifier of the value to be set in the report.

LinkCollection

Optionally specifies a link collection identifier to distinguish between values that share the same usage page and usage identifier. If this parameter is zero, *LinkCollection* is ignored.

Usage

Specifies the usage identifier of the value to be set in the report.

UsageValue

Specifies the data that is to be set in the value for the report provided at *Report*.

PreparsedData

Points to the preparsed data returned for the device when collection information was obtained at initialization.

Report

Points to a caller-allocated buffer for a report that contains the value to be set.

ReportLength

Specifies the length, in bytes, of the buffer provided at *Report*.

Return Value

HidP_SetUsageValue returns a HIDP_XXX status code from the following list:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_REPORT_TYPE

Indicates that the report type value provided at *ReportType* was invalid.

HIDP_STATUS_INVALID_REPORT_LENGTH

Indicates that the report length provided at *ReportLength* does not match the expected report length for a report of the type specified in *ReportType*.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that no value that matches the given usage, usage page, and link collection could be found in this report.

HIDP_STATUS_INCOMPATIBLE_REPORT_ID

Indicates that the value specified by the parameters *Usage, UsagePage,* and optionally *Link-Collection* is known, but cannot be set in the data provided at *Report*. A new report would need to be allocated for this value.

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that the value specified by the parameters *Usage, UsagePage,* and optionally *Link-Collection* could not be set in the report because it is invalid.

Comments

This routine does not automatically handle the sign bit. Callers must either manually set the sign bit, at the position provided in the HIDP_VALUE_CAPS structure for this value, or call **HidP_SetScaledUsageValue**.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_SetScaledUsageValue, HIDP_VALUE_CAPS

HidP_SetUsageValueArray

```
NTSTATUS
  HidP_SetUsageValueArray(
    IN HIDP_REPORT_TYPE ReportType,
    IN USAGE UsagePage,
    IN USHORT LinkCollection    OPTIONAL,
    IN USAGE Usage,
    IN PCHAR UsageValue,
    IN USHORT UsageValueByteLength,
    IN PHIDP_PREPARSED_DATA PreparsedData,
    OUT PCHAR Report,
    IN ULONG ReportLength
    );
```

HidP_SetUsageValueArray sets a series of values into a report for a usage which has more than one data item for the single usage.

Parameters

ReportType

Specifies the type of report provided at *Report*. *ReportType* must be one of the following values:

HidP_Input

Specifies that the report is an input report.

HidP_Output

Specifies that the report is an output report.

HidP_Feature

Specifies that the report is a feature report.

UsagePage

Specifies the usage page identifier of the data items to be set in the report.

LinkCollection

Optionally specifies the link collection identifier of the data items to be set in the report.

Usage

Specifies the usage identifier of the data items to be set in the report.

UsageValue

Points to a caller-allocated buffer that contains the data to be set in the report provided at *Report*. The correct length for this buffer can be found by multiplying the **ReportCount** and **BitSize** fields of the HIDP_VALUE_CAPS structure for this value and rounding the resultant value up to the nearest byte.

UsageValueByteLength

Specifies the length, in bytes, of the buffer provided at *UsageValue*.

PreparsedData

Points to the preparsed data returned for the device when collection information was obtained at initialization.

Report

Points to a caller-allocated buffer for a report to hold the data items for transmission to the device.

ReportLength

Specifies the length, in bytes, of the buffer provided at *Report*.

Return Value

HidP_SetUsageValueArray returns a HIDP_XXX status code from the following list:

HIDP_STATUS_SUCCESS

Indicates that the routine completed successfully.

HIDP_STATUS_INVALID_PREPARSED_DATA

Indicates the preparsed HID device data provided at *PreparsedData* is malformed.

HIDP_STATUS_INVALID_REPORT_TYPE

Indicates that the report type value provided at *ReportType* was invalid.

HIDP_STATUS_INVALID_REPORT_LENGTH

Indicates that the report length provided at *ReportLength* does not match the expected report length for a report of type specified in *ReportType*.

HIDP_STATUS_NOT_VALUE_ARRAY

Indicates that the requested usage has only one data item. To set the data, clients should call **HidP_SetUsageValue** or **HidP_SetScaledUsageValue** instead.

HIDP_STATUS_INCOMPATIBLE_REPORT_ID

Indicates that the data items specified by the parameters *Usage, UsagePage,* and optionally *LinkCollection* is known, but cannot be set in the data provided at *Report*. A new report would need to be allocated for this value.

HIDP_STATUS_USAGE_NOT_FOUND

Indicates that the data items specified by the parameters *Usage, UsagePage,* and optionally *LinkCollection* could not be set in the report because a usage matching those parameters could not be found for this top-level collection.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

HidP_SetScaledUsageValue, **HidP_SetUsageValue**

HidP_TranslateUsagesToI8042ScanCodes

This routine is to be determined.

HidP_UsageListDifference

```
NTSTATUS
  HidP_UsageListDifference(
    IN PUSAGE PreviousUsageList,
    IN PUSAGE CurrentUsageList,
    OUT PUSAGE BreakUsageList,
    OUT PUSAGE MakeUsageList,
    IN ULONG UsageListLength
    );
```

HidP_UsageListDifference compares and provides the differences between two lists of buttons.

Parameters

PreviousUsageList

Points to the older button list to be used for comparison.

CurrentUsageList

Points to the newer button list to be used for comparison.

BreakUsageList

Points to a caller-allocated buffer that, on return, contains the buttons that are set in the older list, provided at *PreviousUsageList*, but not set in the new list, provided at *CurrentUsageList*.

MakeUsageList

Points to a caller-allocated buffer that, on return, contains the buttons that are set in the new list, provided at *CurrentUsageList*, but not set in the old list, provided at *PreviousUsageList*.

UsageListLength

Specifies the length, *in array elements*, of the buffers provided at *CurrentUsageList* and *PreviousUsageList*.

Return Value

HidP_UsageListDifference returns HIDP_STATUS_SUCCESS.

Comments

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

CHAPTER 3

HID Structures for Clients

This chapter describes system-defined structures specific to Human Input Device (HID) drivers and clients. See Part 1 for information about general system-defined structures that are not described here. See Chapter 4 for system structures specific to USB client drivers.

Drivers can use only those members of structures that are described here. All undocumented members of these structures are reserved for system use.

HID_COLLECTION_INFORMATION

```
typedef struct _HID_COLLECTION_INFORMATION {
  ULONG DescriptorSize ;
  BOOLEAN Polled ;
  .
  .
} HID_COLLECTION_INFORMATION, *PHID_COLLECTION_INFORMATION ;
```

Members

DescriptorSize

Is the size, in bytes, required to hold a collection descriptor for this device.

Polled

Indicates that this device is on a non-USB bus and the data is provided in a polled manner.

Comments

This structure is used by kernel mode HID clients to retrieve general information about a top level collection. For detailed information the collection descriptor must be consulted.

HIDP_COLLECTION_DESC

```
typedef struct _HIDP_COLLECTION_DESC
   USAGE   UsagePage;
   USAGE   Usage;
   UCHAR   CollectionNumber;
   UCHAR   Reserved [15]; // Must be zero
   USHORT  InputLength;
   USHORT  OutputLength;
   USHORT  FeatureLength;
   USHORT  PreparsedDataLength;
   PHIDP_PREPARSED_DATA  PreparsedData;
} HIDP_COLLECTION_DESC, *PHIDP_COLLECTION_DESC;
```

HIDP_COLLECTION_DESC is used by HID clients to hold information about a HID collection descriptor.

Members

UsagePage

The page value that, when prefixed to the usage id, defines the usage for the collection.

Usage

The usage id for the collection. A value which, together with **UsagePage,** uniquely describes the use of the collection. .

CollectionNumber

A number assigned to the collection by the HID class driver, and used by the class driver to uniquely identify the collection.

InputLength

Length of all input reports.

OutputLength

Length of all output reports.

FeatureLength

Length of all feature reports.

PreparsedDataLength

Length of the preparsed data pointed to by **PreparsedData**.

PreparsedData

Points to a buffer containing the preparsed report descriptor data for this collection. Individual values can be extracted from the preparsed data by passing this buffer to the kernel

mode HID parser routines (HidP_Xxx) in HIDPARSE.SYS. For further details see *HID Support Routines for Clients*.

Comments

This structure is used by kernel mode HID clients to retrieve the collection descriptor for a top level collection.

The lengths of all input, output, and feature reports are normalized by the collection they belong to. For example, although a collection can have many feature items and, therefore, potentially many feature reports, the length of of a feature report is always **FeatureLength**. **FeatureLength** is the length of the longest feature report belonging to the collection. Feature items that require reports smaller than **FeatureLength** still produce reports of length **FeatureLength**. Unused space in such reports is zero-filled.

Kernel-mode clients must obtain preparsed data using IOCTL_HID_GET_COLLECTION_DESC. The HID support routine **HidD_GetPreparsedData** available to user-mode clients for acquiring preparsed data is not available in kernel mode. User-mode clients must use **HidD_GetPreparsedData** to obtain preparsed data.

HIDD_ATTRIBUTES

```
typedef struct _HIDD_ATTRIBUTES {
  ULONG   Size;
  USHORT  VendorID;
  USHORT  ProductID;
  USHORT  VersionNumber;
} HIDD_ATTRIBUTES, *PHIDD_ATTRIBUTES;
```

HIDD_ATTRIBUTES is used by HID clients to obtain vendor identifying information about a given HID device.

Members

Size

Is the size of the HIDD_ATTRIBUTES structure, in bytes. Before using this structure with a system routine that uses this structure, the caller must set this field to the size of the structure as such:

```
sizeof(HIDD_ATTRIBUTES)
```

VendorID

Specifies the vendor identifier assigned to a given HID device.

ProductID

Specifies the product identifier assigned to a given HID device.

VersionNumber

Specifies the manufacturer's revision number for a given HID device.

Comments

This structure is used by clients when calling **HidD_GetAttributes** to obtain vendor identifying information reported by the device to the system HID components.

See Also

HidD_GetAttributes

HIDD_CONFIGURATION

This structure is to be determined.

HIDP_BUTTON_CAPS

```
typedef struct _HIDP_BUTTON_CAPS
{
  USAGE    UsagePage;
  UCHAR    ReportID;
  .
  .
  USHORT   BitField;
  USHORT   LinkCollection;
  USAGE    LinkUsage;
  USAGE    LinkUsagePage;
  BOOLEAN  IsRange;
  BOOLEAN  IsStringRange;
  BOOLEAN  IsDesignatorRange;
  BOOLEAN  IsAbsolute;
  .
  .
  union
  {
    struct
    {
      USAGE    UsageMin,          UsageMax;
      USHORT   StringMin,         StringMax;
      USHORT   DesignatorMin,     DesignatorMax;
    } Range;
```

```
    struct
    {
      USAGE    Usage ;
      USHORT   StringIndex ;
      USHORT   DesignatorIndex ;
    } NotRange;
  };
} HIDP_BUTTON_CAPS, *PHIDP_BUTTON_CAPS;
```

HIDP_BUTTON_CAPS is used by HID clients to hold the capability data for a button on a HID device.

Members

UsagePage

Specifies the usage page identifier for this button.

ReportID

Specifies the identifier of the report to which this button belongs.

BitField

Is the bitfield, as defined by the HID specification, that describes the behavior of this button. This is the raw data for the bitfield found after the main item in the HID descriptor. The bit meanings from this bitfield are further parsed into BOOLEAN values elsewhere in this structure.

LinkCollection

Specifies a unique identifier to identify the link collection in which this control is contained. A value of zero in this member indicates that the control is contained in the top level collection. This value of this member is also the index into an array of HIDP_LINK_COLLECTION_NODE structures returned from **HidP_GetLinkCollectionNodes**.

LinkUsage

Specifies the usage identifier for the link collection that this button belongs to. If **LinkCollection** is zero, this member is the usage identifier for the top level collection.

LinkUsagePage

Specifies the usage page identifier for the link collection that this button belongs to. If **LinkCollection** is zero, this member is the usage page identifier for the top level collection.

IsRange

Specifies, if TRUE, that this button has a range of usage identifiers that span the values between **Range.UsageMin** and **Range.UsageMax**. If this member is FALSE, the single usage identifier for this button is provided in **NotRange.Usage**.

IsStringRange

Specifies, if TRUE, that this button has a range of string index identifiers that span the values between **Range.StringMin** and **Range.StringMax**. If this member is FALSE, the single string index identifier for this button is provided in **NotRange.StringIndex**.

IsDesignatorRange

Specifies, if TRUE, that this button has a range of designator identifiers that span the values between **Range.DesignatorMin** and **Range.DesignatorMax**. If this member is FALSE, the single designator identifier for this button is provided in **NotRange.DesignatorIndex**.

IsAbsolute

Specifies, if TRUE, that this button provides absolute data and not the change in state from the last value.

Range.UsageMin

Specifies the lower value of a range of usage identifiers that is bounded at the upper edge by **Range.UsageMax**.

Range.UsageMax

Specifies the upper value of a range of usage identifiers that is bounded at the lower edge by **Range.UsageMin**.

Range.StringMin

Specifies the lower value of a range of string identifiers that is bounded at the upper edge by **Range.StringMax**.

Range.StringMax

Specifies the upper value of a range of string identifiers that is bounded at the lower edge by **Range.StringMin**.

Range.DesignatorMin

Specifies the lower value of a range of designator identifiers that is bounded at the upper edge by **Range.DesignatorMax**.

Range.DesignatorMax

Specifies the upper value of a range of designator identifiers that is bounded at the lower edge by **Range.DesignatorMin**.

NotRange.Usage

Specifies the usage identifier for this button.

NotRange.StringIdentifier

Specifies a string index identifier for this button.

NotRange.DesignatorIdentifier

Specifies a designator identifier for this button.

Comments

HIDP_BUTTON_CAPS is used by HID to deliver and hold capability information about buttons for a HID top level collection. To obtain the button capability, clients call **HidP_GetButtonCaps** and **HidP_GetSpecificButtonCaps**. Callers of those routines allocate buffers of HIDP_BUTTON_CAPS structures to be used as a parameter to those routines. The correct length of the buffers required can be found in the HIDP_CAPS structure that is filled in by **HidP_GetCaps**.

See *HIDP_VALUE_CAPS* for information on structures that hold capability information about non-button data.

See Also

HidP_GetButtonCaps, **HidP_GetCaps**, **HidP_GetSpecificButtonCaps**, HIDP_CAPS

HIDP_CAPS

```
typedef struct _HIDP_CAPS {
  USAGE Usage;
  USAGE UsagePage ;
  USHORT InputReportByteLength ;
  USHORT OutputReportByteLength ;
  USHORT FeatureReportByteLength ;
  .
  .
  USHORT NumberLinkCollectionNodes ;
  USHORT NumberInputButtonCaps ;
  USHORT NumberInputValueCaps ;
  USHORT NumberOutputButtonCaps ;
  USHORT NumberOutputValueCaps ;
  USHORT NumberFeatureButtonCaps ;
  USHORT NumberFeatureValueCaps ;
} HIDP_CAPS, *PHIDP_CAPS ;
```

HIDP_CAPS is used by HID clients to hold the capabilities of a HID device.

Members

Usage

Specifies the specific class of functionality that this device provides. This value is dependent and specific to the value provided in **UsagePage**. For example, a keyboard could have a **UsagePage** of HID_USAGE_PAGE_GENERIC and a **Usage** of HID_USAGE_GENERIC_KEYBOARD.

UsagePage

Specifies the usage page identifier for this top-level collection.

InputReportByteLength

Specifies the maximum length, in bytes, of an input report for this device including the report ID which is unilaterally prepended to the device data.

OutputReportByteLength

Specifies the maximum length, in bytes, of an output report for this device including the report ID which is unilaterally prepended to the device data.

FeatureReportByteLength

Specifies the maximum length, in bytes, of a feature report for this device including the report ID which is unilaterally prepended to the device data.

NumberLinkCollectionNodes

Specifies the number of HIDP_LINK_COLLECTION_NODE structures that are returned for this top-level collection by **HidP_GetLinkCollectionNodes**.

NumberInputButtonCaps

Specifies the number of input buttons.

NumberInputValueCaps

Specifies the number of input values.

NumberOutputButtonCaps

Specifies the number of output buttons.

NumberOutputValueCaps

Specifies the number of output values.

NumberFeatureButtonCaps

Specifies the number of feature buttons.

NumberFeatureValueCaps

Specifies the number of feature values.

Comments

HIDP_CAPS holds the parsed capabilities and data maximums returned for a device from **HidP_GetCaps**.

See Also

HidP_GetCaps, **HidP_GetLinkCollectionNodes**

HIDP_LINK_COLLECTION_NODE

```
typedef struct _HIDP_LINK_COLLECTION_NODE {
  USAGE    LinkUsage;
  USAGE    LinkUsagePage;
  USHORT   Parent;
  USHORT   NumberOfChildren;
  USHORT   NextSibling;
  USHORT   FirstChild;
  .
  .
} HIDP_LINK_COLLECTION_NODE, *PHIDP_LINK_COLLECTION_NODE;
```

HIDP_LINK_COLLECTION_NODE structures hold descriptive data about each link collection that is part of a top level collection.

Members

LinkUsage

Specifies the usage identifier for the link collection described by this structure.

LinkUsagePage

Specifies the usage page identifier for the link collection described by this structure.

Parent

Specifies the index, into the array of HIDP_LINK_COLLECTION_NODE structures returned by **HidP_GetLinkCollectionNodes**, of the parent of this link collection. If the link collection has no parent, this value will be zero.

NumberOfChildren

Specifies the number of link collections that are contained within the link collection described by this structure.

NextSibling

Specifies the index, into an array of HIDP_LINK_COLLECTION_NODE structures returned by **HidP_GetLinkCollectionNodes**, of the next link collection that is at the same level in the hierarchy of link collections as this link collection.

FirstChild

Specifies the index, into an array of HIDP_LINK_COLLECTION_NODE structures returned by **HidP_GetLinkCollectionNodes**, of the first link collection that is a child of this collection that is not a sibling of another link collection at the same level.

Comments

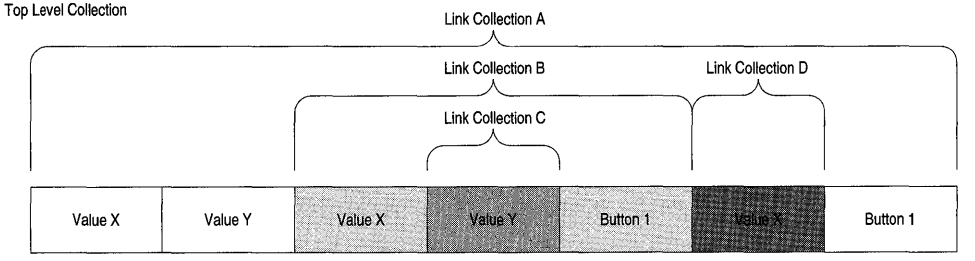


Figure 3.1 Link Collection Nodes

A HIDP_LINK_COLLECTION_NODE structure corresponds to a HID-defined link collection and describes each subcollection in a top-level collection. A top-level collection can have any number of subcollections or none. Defined relationships among link collections depend on their positions in a top level collection as well as their position among other link collections. At most, a link collection can have the following relationships to other link collections within the same top-level collection:

- One parent
- Any number of children
- A sibling
- A first child

A top level collection can have zero or more children, of which only one can be considered a first child. Consider the preceding diagram as an example of a top level collection with link collections.

The top level collection has a total of four link collections contained within it. However, it does not have four children. The top level collection has one child, link collection A.

Link collection A has two children, link collection B and link collection D. Assuming the collection is parsed from left to right in the above diagram, link collection B is encountered first. This makes it the first child of link collection A for the time being. Ignoring for the moment that link collection B has children, link collection D is encountered next. Link

collection B now becomes the sibling of link collection D and D now becomes the first child of A. Siblings are defined as the next link collection encountered that is not a child of a different parent link collection. Link collection C is a child of link collection B. This makes link collection B the parent of C in the same manner as A is the parent of B.

A first child, the member **FirstChild** in a HIDP_LINK_COLLECTION_NODE structure, is a link collection that is the last child encountered for a specific parent link collection and no collection claims it as a sibling. In the preceding example, link collection D is the first child of link collection A because no collection has it as a sibling. In the same manner, link collection C is the first child of B.

The next sibling, the member **NextSibling** in a HIDP_LINK_COLLECTION_NODE structure, is the link collection that just previous in the order of parsing. In the example. the next sibling for link collection D is link collection B. Link collections B and A have no next sibling.

When **HidP_GetLinkCollectionNodes** returns an array of HIDP_LINK_COLLECTION_NODE structures it uses the members of each structure to describe the relationships of the link collections. When an index is returned in a structure it is the index into the array such that it can be used to retrieve the link collection data for that relationship. The link collection index is the same value as is found in the **LinkCollection** member of the HIDP_VALUE_CAPS and HIDP_BUTTON_CAPS structures.

The following code is an example of using a link collection node index to find the first child of link collection seven:

```
HIDP_LINK_COLLECTION_NODE Collection[10] ;
HIDP_LINK_COLLECTION_NODE Node1 ;

Node1 = Collection[Collection.FirstChild[7]] ;
```

See Also

HidP_GetLinkCollectionNodes, HIDP_BUTTON_CAPS, HIDP_VALUE_CAPS

HIDP_VALUE_CAPS

```
typedef struct _HIDP_VALUE_CAPS {
  USAGE    UsagePage;
  UCHAR    ReportID;
  .
  .
  USHORT   BitField;
  USHORT   LinkCollection;
  USAGE    LinkUsage;
  USAGE    LinkUsagePage;
  BOOLEAN  IsRange;
  BOOLEAN  IsStringRange;
```

```
  BOOLEAN  IsDesignatorRange;
  BOOLEAN  IsAbsolute;
  BOOLEAN  HasNull;
  .
  .
  USHORT   BitSize;
  USHORT   ReportCount;
  .
  .
  LONG     LogicalMin,       LogicalMax;
  LONG     PhysicalMin,      PhysicalMax;
  union
  {
  struct
    {
      USAGE    UsageMin,         UsageMax;
      USHORT   StringMin,        StringMax;
      USHORT   DesignatorMin,    DesignatorMax;
    } Range;
  struct
    {
      USAGE    Usage;
      USHORT   StringIndex;
      USHORT   DesignatorIndex;
    } NotRange;
  };
} HIDP_VALUE_CAPS, *PHIDP_VALUE_CAPS;
```

HIDP_VALUE_CAPS is used by HID clients to hold the capability data for a value from a HID device.

Members

UsagePage

Specifies the usage page identifier for this value.

ReportID

Specifies the report ID in which this value is contained.

BitField

Is the bitfield, as defined by the HID specification, that describes the behavior of this value. This is the raw data for the bitfield found after the main item in the HID descriptor. The bit meanings from this bitfield are further parsed into BOOLEAN values elsewhere in this structure.

LinkCollection

Specifies a unique identifier to distinguish two controls, within a collection, that have the same usage page and usage identifiers.

LinkUsage

Specifies the usage identifier for the link collection that this value belongs to. If **LinkCollection** is zero, this member is the usage identifier for the top level collection.

LinkUsagePage

Specifies the usage page identifier for the link collection that this value belongs to. If **LinkCollection** is zero, this member is the usage page identifier for the top level collection.

IsRange

Specifies, if TRUE, that this value has a range of usage identifiers that span the members **Range.UsageMin** and **Range.UsageMax**. If this member is FALSE, the single usage identifier for this value is provided in **NotRange.Usage**.

IsStringRange

Specifies, if TRUE, that this value has a range of string index identifiers that span the members **Range.StringMin** and **Range.StringMax**. If this member is FALSE, the single string index identifier for this value is provided in **NotRange.StringIndex**.

IsDesignatorRange

Specifies, if TRUE, that this value has a range of designator identifiers that span the members **Range.DesignatorMin** and **Range.DesignatorMax**. If this member is FALSE, the single designator identifier for this value is provided in **NotRange.DesignatorIndex**.

IsAbsolute

Specifies, if TRUE, that this value provides absolute data as opposed to the change from the last value.

HasNull

Specifies that this value has a condition where the data in this value is not meaningful. When this occurs, HIDP_STATUS_NULL is returned when the value is retrieved.

BitSize

Specifies the number of bits dedicated within a report to a single instance of this value. If **ReportCount** is greater than one, this bit size is size of each data item, not the size of all data items.

ReportCount

Specifies the number of data items that the usage identifier, described by this structure, contains.

LogicalMin

Specifies the lowest, or minimum, signed number that this value will report.

LogicalMax

Specifies the highest, or maximum, signed number that this value will report.

PhysicalMin

Specifies the lowest, or minimum, signed number that this value will report after scaling is applied to the logical value.

PhysicalMax

Specifies the highest, or maximum, number that this value will report after scaling is applied to the logical value.

Range.UsageMin

Specifies the lower value of a range of usage identifiers that is bounded at the upper edge by **Range.UsageMax**.

Range.UsageMax

Specifies the upper value of a range of usage identifiers that is bounded at the lower edge by **Range.UsageMin**.

Range.StringMin

Specifies the lower value of a range of string identifiers that is bounded at the upper edge by **Range.StringMax**.

Range.StringMax

Specifies the upper value of a range of string identifiers that is bounded at the lower edge by **Range.StringMin**.

Range.DesignatorMin

Specifies the lower value of a range of designator identifiers that is bounded at the upper edge by **Range.DesignatorMax**.

Range.DesignatorMax

Specifies the upper value of a range of designator identifiers that is bounded at the lower edge by **Range.DesignatorMin**.

NotRange.Usage

Specifies the usage identifier for this value.

NotRange.StringIdentifier

Specifies a string index identifier for this value.

NotRange.DesignatorIdentifier

Specifies a designator identifier for this value.

Comments

HIDP_VALUE_CAPS is used by HID to deliver and hold capability information about values in a HID top-level collection. HID clients that use the **HidP_***Xxx* routines allocate an array of these structures as a buffer for **HidP_GetValueCaps** and **HidP_GetSpecificValueCaps** to obtain information about values in a top level collection.

See *HIDP_BUTTON_CAPS* for information on structures that hold capability information about button data.

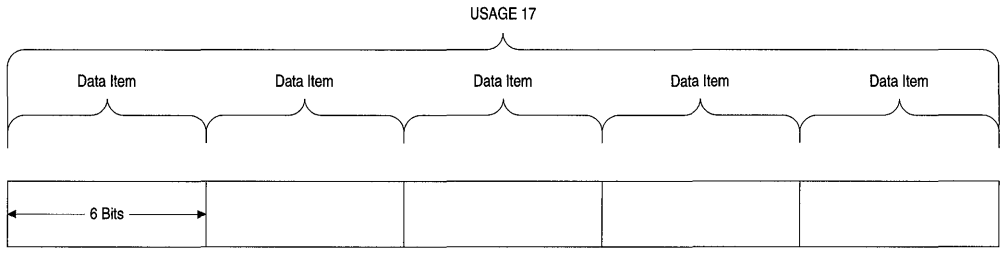


Figure 3.2 Usage Arrays

The member **ReportCount** indicates how many data items are present for each usage that is described by an instance of this structure.

In Figure 3.2, an example usage layout is given. In this case the device has reported that for usage 17, it has five data items each six bits long. **ReportCount** for this structure would equal 5.

If **ReportCount** is equal to one, clients access the data for the usage by calling **HidP_GetUsageValue** or **HidP_GetScaledUsageValue**. However, if **ReportCount** is greater than one, then **HidP_GetUsageValue** and **HidP_GetScaledUsageValue** will not return data except for the first item. Clients should call **HidP_GetUsageValueArray** instead.

See Also

HidP_GetSpecificValueCaps, **HidP_GetUsageValue**, **HidP_GetUsageValueArray**, **HidP_GetValueCaps**

USAGE_AND_PAGE

```
typedef struct _USAGE_AND_PAGE {
  USAGE Usage;
  USAGE UsagePage;
} USAGE_AND_PAGE, *PUSAGE_AND_PAGE;
```

USAGE_AND_PAGE is used by HID clients when obtaining the status of buttons to hold the usage page and usage of a button that is down.

Members

Usage

Specifies the usage identifier within the usage page specified by **UsagePage** of a button that is down.

UsagePage

Specifies the usage page identifier of a button that is down.

Comments

Clients use USAGE_AND_PAGE with **HidP_GetButtonsEx** to obtain both the usage page and usage identifiers of each button that is down.

See Also

HidP_GetButtonsEx

CHAPTER 4

HID Support Routines for MiniDrivers

Human input device (HID) minidrivers can call the following routines to support HID devices.

Routines in this chapter are listed in alphabetic order.

HidRegisterMinidriver

```
NTSTATUS
  HidRegisterMinidriver(
    IN PHID_MINIDRIVER_REGISTRATION MinidriverRegistration
    );
```

HidRegisterMinidriver is called by HID minidrivers, during their initialization, to register with the HID class services.

Parameters

MinidriverRegistration

Points to a caller-allocated buffer that contains an initialized HID_MINIDRIVER_REGISTRATION structure for the minidriver.

Return Value

HidRegisterMinidriver returns one of the following NTSTATUS codes:

STATUS_SUCCESS

Indicates that the routine completed without error and the minidriver is now registered with the HID class driver.

STATUS_INSUFFICIENT_RESOURCES

Indicates that there was insufficient memory for the system to register the minidriver.

STATUS_REVISION_MISMATCH

Indicates that the HID revision number provided in *MinidriverRegistration*->Revision is not supported by this version of the HID class driver.

Comments

Before calling this routine, minidrivers must initialize all fields of the HID_MINIDRIVER_REGISTRATION structure that is provided at *MinidriverRegistration*. See HID_MINIDRIVER_REGISTRATION for details on these fields.

Callers of this routine must be running at IRQL < DISPATCH_LEVEL.

See Also

HID_MINIDRIVER_REGISTRATION

CHAPTER 5

HID Structures for Minidrivers

This chapter describes system-defined structures specific to Windows® Driver Model (WDM) Human Input Device (HID) minidrivers. See Part 1 for information about general operating system-defined structures that are not described here. See *HID Structures for Clients* for system structures specific to USB client drivers.

Drivers can use only those members of structures that are described here. All undocumented members of these structures are reserved for system use.

Structures described in this chapter are in alphabetical order.

HID_DEVICE_ATTRIBUTES

```
typedef struct _HID_DEVICE_ATTRIBUTES {
  ULONG           Size;
  USHORT          VendorID;
  USHORT          ProductID;
  USHORT          VersionNumber;
  .
  .
} HID_DEVICE_ATTRIBUTES, * PHID_DEVICE_ATTRIBUTES;
```

HID_DEVICE_ATTRIBUTES is the HID class driver-defined structure used to hold information about a HID device.

Members

Size

Contains the size of the structure. This field should be treated as read-only when using this structure to complete an IOCTL_HID_GET_DEVICE_ATTRIBUTES request.

VendorID

Specifies the manufacturer-determined identification value for the HID device.

ProductID

Specifies the manufacturer-determined product identification value.

VersionNumber

Specifies the manufacturer-determined version number for the HID device.

Comments

HID_DEVICE_ATTRIBUTES is typically used in conjunction with an I/O control request to a miniclass driver to obtain the attribute information. Miniclass drivers fill in the **VendorID**, **ProductID**, and **VersionNumber** members of this structure to complete an IOCTL_HID_GET_DEVICE_ATTRIBUTES request from the system-supplied class driver.

HID_DEVICE_EXTENSION

```
typedef struct _HID_DEVICE_EXTENSION {
  PDEVICE_OBJECT  PhysicalDeviceObject;
  PDEVICE_OBJECT  NextDeviceObject;
  PVOID           MiniDeviceExtension;
} HID_DEVICE_EXTENSION, *PHID_DEVICE_EXTENSION;
```

HID_DEVICE_EXTENSION is the HID class-defined structure that is used by all minidrivers for their device extension in their functional device object.

Members

PhysicalDeviceObject

Points to the physical device object for this HID minidriver.

NextDeviceObject

Points to the next device object in the chain of drivers of which the minidriver is a part.

MiniDeviceExtension

Points to the minidriver-specific portion of the device extension.

Comments

A HID minidriver functional device object's device extension will always be in a format corresponding to this structure. When the device extension is initialized for the device object, the members of this structure will be initialized by the class driver. The minidriver can use the memory pointed to by **MiniDeviceExtension** for its own device-specific data area. Minidrivers should only reference and not change any of the data except in the memory at **MiniDeviceExtension**.

When a miniclass driver sends an IRP to a lower driver in its chain, it should use the device object at **NextDeviceObject** as the target device object in a call to **IoCallDriver**. The device object pointed to by **PhysicalDeviceObject** should be used as a reference only, as it points to the PDO for this device, not necessarily to the next device object in the chain of drivers. Other drivers may be layered above the physical device object pointed to by **PhysicalDeviceObject**. As such, **NextDeviceObject** will always point to the top of the stack that the minidriver should send its IRPs to.

HID_MINIDRIVER_REGISTRATION

```
typedef struct _HID_MINIDRIVER_REGISTRATION {
  ULONG           Revision;
  PDRIVER_OBJECT  DriverObject;
  PUNICODE_STRING RegistryPath;
  ULONG           DeviceExtensionSize;
  BOOLEAN         DevicesArePolled;
  .
  .
} HID_MINIDRIVER_REGISTRATION, *PHID_MINIDRIVER_REGISTRATION;
```

HID_MINIDRIVER_REGISTRATION is used by HID minidrivers to describe the minidriver when registered with the HID class driver.

Members

Revision

Specifies the HID version that this minidriver supports.

DriverObject

Points to the DRIVER_OBJECT for the minidriver.

RegistryPath

Points to the registry path for the minidriver.

DeviceExtensionSize

Specifies the length, in bytes, that the minidriver requests for a device extension.

DevicesArePolled

Specifies that the devices on the bus that this minidriver supports must be polled in order to obtain data from the device.

Comments

Miniclass drivers fill in an instance of this structure to be used as a parameter to **HidRegisterMinidriver**. The structure must be zero-initialized before clients set members of

this structure. Clients should set the members **DriverObject** and **RegistryPath** to the driver object and registry path parameters that are passed to the minidriver as system-supplied parameters to its **DriverEntry** routine. **Revision** should be set to HID_REVISION.

See Also

HidRegisterMinidriver

HID_XFER_PACKET

```
ypedef struct _HID_XFER_PACKET {
  PUCHAR  reportBuffer;
  ULONG   reportBufferLen;
  UCHAR   reportId;
} HID_XFER_PACKET, *PHID_XFER_PACKET;
```

This structure is used by HID miniclass drivers to obtain the parameters and settings for data transfer requests from the class driver.

Members

reportBuffer

Points to a buffer that is provided to the miniclass driver to transfer data.

reportBufferLen

Specifies the length of the buffer at **reportBuffer**.

reportId

Optionally specifies the report ID of the report contained at **reportBuffer**.

Comments

This structure is used by the HID class driver to submit write report and set feature requests to HID miniclass drivers. For information on when this structure is used, see *I/O Requests Serviced by HID Minidrivers* in Chapter 1.

CHAPTER 6

Kbdclass Driver Reference

This chapter includes the following topics about *Kbdclass*, the Microsoft® Windows® 2000 system class driver for device class GUID_CLASS_KEYBOARD:

- *Kbdclass Major I/O Requests*
- *Kbdclass Device Control Requests*
- *Kbdclass Class Service Callback Routine*

Windows 2000 uses Kbdclass as the class driver for all keyboard devices installed in a system. The Windows 2000 Win32® subsystem opens all keyboard devices for its exclusive use. Applications cannot open the keyboard devices operated by Kbdclass.

Kbdclass can work in combination with an optional upper-level keyboard filter driver for a PS/2-style keyboard device. *Kbfiltr*, a sample upper-level keyboard filter driver in the Windows 2000 DDK, demonstrates how to customize the operation of a keyboard device.

For more information on Kbdclass, see the following topics:

- *Keyboard and Mouse Drivers for Non-HID Devices* in the online DDK
- Include file *ntddkbd.h* in the *%user's install path%\inc* directory in the Windows 2000 DDK
- Sample code in the *%user's install path%\src\input* directory in the Windows 2000 DDK

Note

Kbdclass supports legacy devices and Plug and Play devices. As appropriate, this material distinguishes between the operation of Kbdclass for a legacy device and a Plug and Play device. If no distinction is made, the description applies to both legacy and Plug and Play devices.

Kbdclass Major I/O Requests

This section describes the Kbdclass-specific operation of the following major I/O requests that Kbdclass supports:

IRP_MJ_CLEANUP
IRP_MJ_CLOSE
IRP_MJ_CREATE
IRP_MJ_DEVICE_CONTROL
IRP_MJ_FLUSH_BUFFERS
IRP_MJ_INTERNAL_DEVICE_CONTROL
IRP_MJ_PNP
IRP_MJ_POWER
IRP_MJ_READ
IRP_MJ_SYSTEM_CONTROL

For more information on the generic operation of these requests, see *IRP Function Codes and IOCTLs* in Part 1.

IRP_MJ_CLOSE

Operation

The IRP_MJ_CLOSE request closes a keyboard device.

Plug and Play Operation

Kbdclass sends the request down the driver stack and clears the file that is permitted read access to the device. If there is a grandmaster device, Kbdclass sends a close request to all the function device objects that are associated with the subordinate class device objects.

Legacy Operation

Kbdclass sends an IOCTL_INTERNAL_KEYBOARD_DISABLE request to the port driver. Kbdclass also clears the file that is permitted read access to the device.

I/O Status Block

Plug and Play Operation

The **Information** member is set to zero.

The **Status** member is set to STATUS_SUCCESS or to the status returned by the function driver for the IRP_MJ_CLOSE request.

Legacy Operation

The **Information** member is set to zero.

The **Status** member is set to STATUS_SUCCESS or to the status returned by the port driver for the IOCTL_INTERNAL_KEYBOARD_DISABLE request.

IRP_MJ_CREATE

Operation

The IRP_MJ_CREATE request opens a file on a keyboard device.

Plug and Play Operation

If the device has been started, Kbdclass sends the IRP_MJ_CREATE request down the driver stack. If the device is not started, Kbdclass completes the request without sending the request down the driver stack. Kbdclass sets the trusted file that is permitted read access to the device. If there is a grandmaster device, Kbdclass sends a create request to all the keyboard devices that are associated with the subordinate class device objects.

Legacy Operation

Kbdclass sends a synchronous IOCTL_INTERNAL_KEYBOARD_ENABLE request down the device stack.

I/O Status Block

Plug and Play Operation

The **Information** member of the IRP is set to zero.

The **Status** member is set to one of the following values:

- **STATUS_SUCCESS**
- **STATUS_UNSUCCESFUL** The device is not started.
- **STATUS_Xxx** A lower-level driver returns an error status.

Legacy Operation

The **Information** member is set to zero.

The **Status** member is set to STATUS_SUCCESS or to the status returned by the port driver for an IOCTL_INTERNAL_KEYBOARD_ENABLE request.

IRP_MJ_DEVICE_CONTROL

Kbdclass supports the following device control requests:

IOCTL_KEYBOARD_QUERY_ATTRIBUTES
IOCTL_KEYBOARD_QUERY_INDICATOR_TRANSLATION
IOCTL_KEYBOARD_QUERY_INDICATORS
IOCTL_KEYBOARD_QUERY_TYPEMATIC
IOCTL_KEYBOARD_SET_INDICATORS

For most other device control requests, Kbdclass skips the current IRP stack location and sends the request down the stack to be completed by a lower-level driver. If Kbdclass does not support the request, it completes the request with a status of STATUS_INVALID_DEVICE_REQUEST.

For more information on device control requests, see *Kbdclass Device Control Requests.*

IRP_MJ_FLUSH_BUFFERS

Operation

The IRP_MJ_FLUSH_BUFFERS request clears the internal data queue.

I/O Status Block

The **Information** member is zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_NOT_SUPPORTED

The target device is associated with a subordinate class device that does not support flushing the internal data queue.

IRP_MJ_INTERNAL_DEVICE_CONTROL

Operation

Kbdclass skips the current IRP stack location and sends the internal device control request down the device stack to be completed by a lower-level driver.

IRP_MJ_PNP

Kbdclass processes the following Plug and Play requests:

IRP_MN_QUERY_PNP_DEVICE_STATE
IRP_MN_QUERY_REMOVE_DEVICE
IRP_MN_QUERY_STOP_DEVICE
IRP_MN_REMOVE_DEVICE
IRP_MN_START_DEVICE
IRP_MN_SURPRIZE_REMOVAL
IRP_MN_STOP_DEVICE

For all other Plug and Play requests, Kbdclass copies the current IRP stack location and sends the request down the device stack without further processing.

For more information on the generic operation of these requests, see *Plug and Play IRPs* in Volume 1 of the *Windows 2000 Driver Development Reference.*

I/O Status Block

Under normal operation the status block values are specific to the minor function request. If an IRP_MJ_PNP request is sent in error to a legacy device, Kbdclass completes the request with a status of STATUS_NOT_SUPPORTED.

IRP_MJ_POWER

Kbdclass supports the following power requests:

IRP_MN_SET_POWER
IRP_MN_QUERY_POWER
IRP_MN_WAIT_WAKE

For all other power requests, Kbdclass copies the current IRP stack location, requests the next power request, and sends the request down the device stack.

For more information on the generic operation of these requests, see *I/O Request for Power Management* in Volume 1 of the *Windows 2000 Driver Development Reference.*

Status I/O Block

Under normal operation the status block values are specific to the minor function request. If a power request is sent in error to a grandmaster device or a legacy device, Kbdclass completes the request with a status of STATUS_NOT_SUPPORTED.

IRP_MJ_READ

Operation

The IRP_MJ_READ request transfers zero or more KEYBOARD_INPUT_DATA structures from Kbdclass's internal data queue to the Win32 subsystem buffer. If there is no data in the data queue, a read request remains pending until it is completed or canceled.

Kbdclass completes a read request and does not send the request down the device stack.

A read request can be canceled. If a cleanup is in progress when a read request is received, no action is taken.

Note that a read request can be completed successfully only if the request was made by a trusted subsystem. Kbdclass performs a privilege check to enforce this restriction. The Win32 subsystem is currently the only trusted subsystem.

Input

Parameters.Read.Length member specifies the size in bytes of zero or more KEYBOARD_INPUT_DATA structures:

```
typedef struct KEYBOARD_INPUT_DATA {
    USHORT UnitId;   // zero-based unit number of the keyboard port
    USHORT MakeCode; // the make scan code (key depression)
    USHORT Flags;    // indicates a break (key release) and
                     // other scan-code info
    USHORT Reserved;
    ULONG ExtraInformation; // device-specific additional
                            // information for the event
} KEYBOARD_INPUT_DATA, *PKEYBOARD_INPUT_DATA;
```

Output

The **AssociatedIrp.SystemBuffer** member points to the output buffer that is allocated by the Win32 subsystem to output the requested number of KEYBOARD_INPUT_DATA structures.

I/O Status Block

The **Information** member specifies the number of bytes that are transferred to the Win32 subsystem output buffer. The number of bytes that are transferred is the smallest of the requested number of bytes and the number of bytes currently in the internal data queue.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

At least one KEYBOARD_INPUT_DATA structure was transferred.

STATUS_BUFFER_TOO_SMALL

The number of requested bytes is not an integer multiple of the size in bytes of a KEYBOARD_INPUT_DATA structure.

STATUS_PRIVILEGE_NOT_HELD

The requesting subsystem does not have read privileges.

STATUS_CANCELLED

The request was canceled before the transfer actually took place.

IRP_MJ_SYSTEM_CONTROL

Kbdclass supports the following Windows Management Instrumentation (WMI) system control requests:

IRP_MN_REGINFO
IRP_MN_QUERY_DATA_BLOCK

IRP_MN_CHANGE_SINGLE_INSTANCE
IRP_MN_CHANGE_SINGLE_ITEM

For all other system control requests, Kbdclass skips the current IRP stack location and sends the request down the device stack without further processing.

Kbdclass calls **WmiSystemControl** to process WMI requests. Kbdclass registers two types of WMI data blocks: one type holds a pointer to a class device object, the other indicates whether or not the class device supports a *wait/wake* operation. Wait/wake operation can be enabled or disabled. If wait/wake operation is supported and enabled, the device wakes the system after a wake event occurs. If wait/wake operation is not supported or disabled, the device cannot wake the system.

When Called

Kbdclass updates WMI registration information after the device is started and removed. The driver calls the **IoWMIRegistrationControl** routine to update WMI registration information.

At the request of a WMI client, the WDM provider sends one of the following requests:

- An IRP_MN_QUERY_DATA_BLOCK request to obtain data in one of Kbdclass's data blocks.
- An IRP_MN_CHANGE_SINGLE_*Xxx* request to change the data in one of Kbdclass's data blocks.

I/O Status Block

If the request is handled by the driver, the **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_DEVICE_REQUEST

STATUS_WMI_GUID_NOT_FOUND

The requested data block is not valid.

STATUS_WMI_INSTANCE_NOT_FOUND

The WMI context is not valid.

If the request is not handled by the driver and sent down the stack, the **Status** member is set by a lower-level driver.

Kbdclass Device Control Requests

This section describes the following device control requests that Kbdclass supports:

IOCTL_KEYBOARD_QUERY_ATTRIBUTES
IOCTL_KEYBOARD_QUERY_INDICATOR_TRANSLATION
IOCTL_KEYBOARD_QUERY_INDICATORS
IOCTL_KEYBOARD_QUERY_TYPEMATIC
IOCTL_KEYBOARD_SET_INDICATORS
IOCTL_KEYBOARD_SET_TYPEMATIC

Kbdclass changes these device control requests into internal device control request and sends the changed request down the device stack.

Kbdclass also sends a number of other device control requests for Plug and Play devices down the device stack without changing the request to an internal device control request. For a list of these requests, see *IOCTL_KEYBOARD_SET_TYPEMATIC*.

Filter drivers between the class and function drivers can filter device control requests either before they send the request down the device stack or after the lower-level drivers complete the request.

If Kbdclass does not support the request, it completes the request with a status of STATUS_INVALID_DEVICE_REQUEST.

Input and Output

The **Parameters.DeviceIoControlCode.IoControlCode** member specifies the control code.

The **AssociatedIrp->SystemBuffer** member is used for request-specific input and output.

The **Parameters.DeviceIoControl.InputBufferLength** member is used to input the request-specific size of the input buffer.

The **Parameters.DeviceIoControl.OutputBufferLength** member is used to input the request-specific size of the output buffer.

I/O Status Block

Usually, the **Information** member is set to zero or to the number of bytes returned in the output buffer. **Information** can also return request-specific values or pointers.

The **Status** member is set to a request-specific value. **Status** is set to STATUS_INVALID_DEVICE_REQUEST if Kbdclass does not support the request.

IOCTL_KEYBOARD_QUERY_ATTRIBUTES

Operation

The IOCTL_KEYBOARD_QUERY_ATTRIBUTES request returns information about the keyboard attributes.

Kbdclass copies the current stack location, sets the **MajorFunction** member of the new stack location to IRP_MJ_INTERNAL_DEVICE_CONTROL, and sends this request down the device stack.

For more information on this request, see the description of this request in *I8042prt Keyboard Internal Device Control Requests* in Chapter 8.

Input

The **Parameters.DeviceIoControl.InputBufferLength** member is set to zero or a value greater than or equal to the size in bytes of a KEYBOARD_UNIT_ID_PARAMETER. A value of zero specifies a default unit ID of zero.

The **AssociatedIrp.SystemBuffer** member points to a client-allocated buffer that is used to input and output information. On input, **AssociatedIrp.SystemBuffer** points to a KEYBOARD_UNIT_ID_PARAMETER structure. The client sets the **UnitId** member of the input structure.

The **Parameters.DeviceIoControl.OutputBufferLength** member specifies the size in bytes of the output buffer, which must be greater than or equal to the size in bytes of a KEYBOARD_ATTRIBUTES structure.

Output

AssociatedIrp.SystemBuffer points to a client-allocated buffer that the lower-level drivers use to output a KEYBOARD_ATTRIBUTES structure.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of a KEYBOARD_ATTRIBUTES structure, otherwise **Information** is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

The **UnitId** value is not valid.

STATUS_BUFFER_TOO_SMALL

The value of **Parameters.DeviceIoControl.InputBufferLength** or **Parameters.DeviceIoControl.OutputBufferLength** is not valid.

IOCTL_KEYBOARD_QUERY_INDICATORS

Operation

The IOCTL_KEYBOARD_QUERY_INDICATORS request returns information on the keyboard indicators.

Kbdclass copies the current stack location, sets the **MajorFunction** member of the new stack location to IRP_MJ_INTERNAL_DEVICE_CONTROL, and sends this request down the device stack.

Input

The **Parameters.DeviceIoControl.InputBufferLength** member is set to zero or a value greater than or equal to the size in bytes of a KEYBOARD_UNIT_ID_PARAMETER. A value of zero specifies a default unit ID of zero.

The **AssociatedIrp.SystemBuffer** member points to a client-allocated buffer that is used to input and output information. On input, **AssociatedIrp.SystemBuffer** points to a KEYBOARD_UNIT_ID_PARAMETER structure. The client sets the **UnitId** member of the input structure.

The **Parameters.DeviceIoControl.OutputBufferLength** member specifies the size in bytes of the output buffer, which must be greater than or equal to the size in bytes of a KEYBOARD_INDICATOR_PARAMETERS structure.

Output

AssociatedIrp.SystemBuffer points to a client-allocated buffer that the lower-level drivers use to output a KEYBOARD_INDICATOR_PARAMETERS structure.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of a KEYBOARD_INDICATOR_PARAMETERS structure.

The **Status** member is set to one the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

The **UnitId** value is not valid.

STATUS_BUFFER_TOO_SMALL

The output buffer cannot hold the KEYBOARD_INDICATOR_PARAMETERS structure.

IOCTL_KEYBOARD_QUERY_INDICATOR_TRANSLATION

Operation

The IOCTL_KEYBOARD_QUERY_INDICATOR_TRANSLATION request returns information on the mapping between scan codes and indicators.

Kbdclass copies the current stack location, sets the **MajorFunction** member of the new stack location to IRP_MJ_INTERNAL_DEVICE_CONTROL, and sends this request down the device stack.

Input

The **Parameters.DeviceIoControl.InputBufferLength** member is set to zero or a value greater than or equal to the size in bytes of a KEYBOARD_UNIT_ID_PARAMETER. A value of zero specifies a default unit ID of zero.

The **AssociatedIrp.SystemBuffer** member points to a client-allocated buffer that is used to input and output information. On input, **AssociatedIrp.SystemBuffer** points to a KEYBOARD_UNIT_ID_PARAMETER structure. The client sets the **UnitId** member of the input structure.

The **Parameters.DeviceIoControl.OutputBufferLength** member specifies the size in bytes of a client-allocated output buffer, which must be greater than or equal to the size in bytes of a KEYBOARD_INDICATOR_TRANSLATION structure. (Note that this structure includes an array of INDICATOR_LIST members and that the number of members in this array is device-specific.)

Output

AssociatedIrp.SystemBuffer points to a client-allocated buffer that the lower-level drivers use to output a KEYBOARD_INDICATOR_TRANSLATION structure.

I/O Status Block

If the request is successful, the **Information** member is set to the number of bytes of translation data in the KEYBOARD_INDICATOR_TRANSLATION structure.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

The UnitId value is not valid.

STATUS_BUFFER_TOO_SMALL

The output buffer cannot hold the KEYBOARD_INDICATOR_TRANSLATION data.

STATUS_NOT_SUPPORTED

The target device is associated with a subordinate class device.

IOCTL_KEYBOARD_QUERY_TYPEMATIC

Operation

The IOCTL_KEYBOARD_QUERY_TYPEMATIC request returns the typematic settings.

Kbdclass copies the current stack location, sets the **MajorFunction** member of the new stack location to IRP_MJ_INTERNAL_DEVICE_CONTROL, and sends this request down the device stack.

Input

The **Parameters.DeviceIoControl.InputBufferLength** member is set to zero or a value greater than or equal to the size in bytes of a KEYBOARD_UNIT_ID_PARAMETER. A value of zero specifies a default unit ID of zero.

The **AssociatedIrp.SystemBuffer** member points to a client-allocated buffer that is used to input and output information. On input, **AssociatedIrp.SystemBuffer** points to a KEYBOARD_UNIT_ID_PARAMETER structure. The client sets the **UnitId** member of the input structure.

Output

AssociatedIrp.SystemBuffer points to the client-allocated buffer that the lower-level drivers use to output a KEYBOARD_TYPEMATIC_PARAMETERS structure.

I/O Status Block

If the request is successful, the **Information** member is set to the number of bytes of a KEYBOARD_TYPEMATIC_PARAMETERS structure.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

The UnitId value is not valid.

STATUS_BUFFER_TOO_SMALL

The output buffer cannot hold the KEYBOARD_TYPEMATIC_PARAMETERS data.

IOCTL_KEYBOARD_SET_INDICATORS

Operation

The IOCTL_KEYBOARD_SET_INDICATORS request sets the keyboard indicators.

Kbdclass copies the current stack location, sets the **MajorFunction** member of the new stack location to IRP_MJ_INTERNAL_DEVICE_CONTROL, and sends this request down the driver stack.

If there is a grandmaster device, Kbdclass normally sets the keyboard indicators of all the subordinate class devices to a global setting. This operation is controlled by the registry entry value **SendOutputToAllPorts** under the key **HKLM\Services\CurrentControlSet\Kbdclass\Parameters**. If **SendOutputToAllPorts** is nonzero, Kdbclass sets all subordinate class devices to a gobal setting. Otherwise, Kbdclass sets only the device whose unit ID is zero.

Input

The **Parameters.DeviceIoControl.InputBufferLength** member specifies the size in bytes of a KEYBOARD_INDICATOR_PARAMETER structure.

The **AssociatedIrp.SystemBuffer** member points to a client-allocated KEYBOARD_INDICATOR_PARAMETERS structure. The client sets the **UnitId** and **LedFlags** members.

I/O Status Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

The value of **Parameters.DeviceIoControl.InputBufferLength** is less than the size in bytes of a KEYBOARD_INDICATOR_PARAMETER structure.

STATUS_INVALID_PARAMETER

The UnitId value is invalid.

STATUS_IO_TIMEOUT

The requested operation timed out on the device.

IOCTL_KEYBOARD_SET_TYPEMATIC

Operation

The IOCTL_KEYBOARD_SET_TYPEMATIC request sets the typematic parameters.

Kbdclass copies the current stack location, sets the **MajorFunction** member of the new stack location to IRP_MJ_INTERNAL_DEVICE_CONTROL, and sends this request down the device stack.

Note that if there is a grandmaster device, Kbdclass normally sets the typematic settings of all the subordinate class devices to the same global setting. This operation is controlled by the grandmaster's registry entry value **SendOutputToAllPorts** under the key **HKLM\Services\CurrentControlSet\Kbdclass\Parameters**.

Input

The **Parameters.DeviceIoControl.InputBufferLength** member is set to a value greater than or equal to the size in bytes of a KEYBOARD_TYPEMATIC_PARAMETERS structure.

The **AssociatedIrp.SystemBuffer** member points to a client-allocated KEYBOARD_TYPEMATIC_PARAMETERS structure. The client sets the **UnitId**, **Rate,** and **Delay** member values.

I/O Status Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of a KEYBOARD_TYPEMATIC_PARAMETERS structure.

STATUS_INVALID_PARAMETER

The UnitId value is invalid.

STATUS_IO_TIMEOUT

The operation timed out.

IOCTL_Xxx

The following control requests are not changed to internal device control requests. Kbdclass skips the current IRP stack location and sends these requests down the device stack to be completed by the lower-level drivers:

IOCTL_GET_SYS_BUTTON_CAPS
IOCTL_GET_SYS_BUTTON_EVENT
IOCTL_HID_GET_DRIVER_CONFIG
IOCTL_HID_SET_DRIVER_CONFIG
IOCTL_HID_GET_POLL_FREQUENCY_MSEC
IOCTL_HID_SET_POLL_FREQUENCY_MSEC
IOCTL_GET_NUM_DEVICE_INPUT_BUFFERS
IOCTL_SET_NUM_DEVICE_INPUT_BUFFERS
IOCTL_HID_GET_COLLECTION_INFORMATION
IOCTL_HID_GET_COLLECTION_DESCRIPTOR
IOCTL_HID_FLUSH_QUEUE
IOCTL_HID_SET_FEATURE

IOCTL_HID_GET_FEATURE
IOCTL_GET_PHYSICAL_DESCRIPTOR
IOCTL_HID_GET_HARDWARE_ID
IOCTL_HID_GET_MANUFACTURER_STRING
IOCTL_HID_GET_PRODUCT_STRING
IOCTL_HID_GET_SERIALNUMBER_STRING
IOCTL_HID_GET_INDEXED_STRING

Kbdclass Class Service Callback Routine

This section describes **KeyboardClassServiceCallback**, the keyboard class service callback routine.

Kbdclass uses an IOCTL_INTERNAL_KEYBOARD_CONNECT request to connect its class service callback to a keyboard device.

KeyboardClassServiceCallback

```
VOID
  KeyboardClassServiceCallback (
    IN PDEVICE_OBJECT DeviceObject,
    IN PKeyboard_INPUT_DATA InputDataStart,
    IN PKeyboard_INPUT_DATA InputDataEnd,
    IN OUT PULONG InputDataConsumed
    );
```

The **KeyboardClassServiceCallback** routine is the class service callback that is provided by Kbdclass. A function driver calls the class service callback in its ISR dispatch completion routine. The class service callback transfers input data from the input data buffer of a device to the class data queue.

Parameters

DeviceObject

Pointer to the class device object.

InputDataStart

Pointer to the first keyboard data packet (KEYBOARD_INPUT_DATA structure) in the port device's input buffer.

InputDataEnd

Pointer to the keyboard data packet that immediately follows the last data packet in the port device's input data buffer.

InputDataConsumed

Pointer to the number of keyboard data packets that are transferred by the routine.

Include

kbdclass.h

Comments

KeyboardClassServiceCallback transfers input data from the input buffer of the device to the class data queue. This routine is called by the ISR dispatch completion routine of the function driver.

KeyboardClassServiceCallback can be supplemented by a filter service callback that is provided by an upper-level keyboard filter driver. A filter service callback filters the keyboard data that is transferred to the class data queue. For example, the filter service callback can delete, transform, or insert data. Kbfiltr, the sample filter driver in the Microsoft® Windows® 2000 DDK, includes **KbFilter_ServiceCallback**, which is a template for a keyboard filter service callback.

KeyboardClassServiceCallback runs in kernel mode at IRQL DISPATCH_LEVEL.

CHAPTER 7

Mouclass Driver Reference

This chapter includes the following topics about *Mouclass*, the Microsoft® Windows® 2000 system class driver for device class GUID_CLASS_MOUSE:

- *Mouclass Major I/O Requests*
- *Mouclass Device Control Requests*
- *Mouclass Class Service Callback Routine*

Windows 2000 uses Mouclass as the class driver for all mouse devices installed in a system. The Windows 2000 Win32® subsystem opens all mouse devices for its exclusive use. Applications can not open the mouse devices opened by Mouclass.

Mouclass can work in combination with an optional upper-level mouse filter driver for a PS/2-style mouse device. *Moufiltr*, a sample upper-level mouse filter driver in the Windows 2000 DDK, can be used to customize the operation of a mouse device.

For more information on Mouclass, see the following topics:

- *Keyboard and Mouse Drivers for Non-HID Devices* in the online DDK
- Include file *ntddkbd.h* in the *%user's install path%\inc* directory of the Windows 2000 DDK
- Sample code in the *%user's install path%\src\input* directory of the Windows 2000 DDK

Note

Mouclass supports legacy devices and Plug and Play devices. When appropriate, this material distinguishes between the operation of Mouclass for a legacy device and a Plug and Play device. If no distinction is made, the description applies to both legacy and Plug and Play devices.

Mouclass Major I/O Requests

This section describes Mouclass-specific operation of the following major I/O requests:

IRP_MJ_CLEANUP
IRP_MJ_CLOSE
IRP_MJ_CREATE
IRP_MJ_DEVICE_CONTROL
IRP_MJ_FLUSH_BUFFERS
IRP_MJ_INTERNAL_DEVICE_CONTROL
IRP_MJ_PNP
IRP_MJ_POWER
IRP_MJ_READ
IRP_MJ_SYSTEM_CONTROL

For more information about how Mouclass handles the generic operation of these requests, see *IRP Function Codes and IOCTLs* in Part 1 of this volume.

IRP_MJ_CLOSE

Plug and Play Operation

Mouclass sends the request down the device stack and clears the "trusted" file that is permitted read access to the device. If there is a grandmaster device, Mouclass sends a request to close to all the ports associated with the subordinate class devices.

Legacy Operation

Mouclass sends an IOCTL_INTERNAL_MOUSE_DISABLE request to the port driver. Mouclass also clears the "trusted" file that is permitted read access to the device.

I/O Status Block

Plug and Play Operation

The **Information** member is set to zero.

Mouclass sets the **Status** member to STATUS_SUCCESS or to the status returned by the function driver for the IRP_MJ_CLOSE request.

Legacy Operation

The **Information** member is set to zero.

The **Status** member is set to STATUS_SUCCESS or to the status returned by the port driver for the IOCTL_INTERNAL_MOUSE_DISABLE request.

IRP_MJ_CREATE

Operation

The IRP_MJ_CREATE request opens a file on mouse device.

Plug and Play Operation

If the device is started, Mouclass sends the IRP_MJ_CREATE request down the device stack. If the device is not started, Mouclass completes the request without sending the request down the driver stack. Mouclass sets the "trusted" file that is permitted read access to the device. If there is a grandmaster device, Mouclass sends a create request to all the mouse devices associated with the subordinate class device objects.

Legacy Operation

Mouclass sends a synchronous IOCTL_INTERNAL_MOUSE_ENABLE request down the device stack.

I/O Status Block

Plug and Play Operation

The **Information** member of the IRP is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_UNSUCCESFUL

The device is not started.

STATUS_Xxx

A lower-level driver returned an error status.

Legacy Operation

The **Information** member is set to zero.

The **Status** member is set to STATUS_SUCCESS or the status returned by a lower-level driver for the IOCTL_INTERNAL_MOUSE_ENABLE request.

IRP_MJ_DEVICE_CONTROL

Mouclass supports the following major device control request:

IOCTL_MOUSE_QUERY_ATTRIBUTES

For most other device control requests, Mouclass skips the current IRP stack location and sends the request down the device stack to be completed by a lower-level driver. If

Mouclass does not support the request, Mouclass completes the request with a status of STATUS_INVALID_DEVICE_REQUEST.

For more information on device control requests, see *Mouclass Device Control Requests*.

IRP_MJ_FLUSH_BUFFERS

Operation

The IRP_MJ_FLUSH_BUFFERS request clears the internal data queue.

I/O Status Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_NOT_SUPPORTED

The target device is associated with a subordinate class device that does not support flushing the internal data queue.

STATUS_PRIVILEGE_NOT_HELD

IRP_MJ_INTERNAL_DEVICE_CONTROL

Operation

Mouclass skips the current IRP stack location and sends the internal device control request down the device stack to be completed by a lower-level driver.

IRP_MJ_PNP

Operation

Mouclass processes the following Plug and Play requests:

IRP_MN_REMOVE_DEVICE
IRP_MN_START_DEVICE
IRP_MN_STOP_DEVICE

For all other Plug and Play requests, Mouclass copies the current IRP stack location and sends the request down the device stack without further processing.

For more information on the generic operation of these requests, see *Plug and Play IRPs* in Volume 1 of the *Windows 2000 Driver Development Reference*.

I/O Status Block

Under normal operation the status block values are specific to the minor function request. If a Plug and Play request is sent in error to a legacy device, Mouclass completes the request with a status of STATUS_NOT_SUPPORTED.

IRP_MJ_POWER

Operation

Mouclass supports the following power requests and, as appropriate, sends them down the device stack:

IRP_MN_SET_POWER
IRP_MN_QUERY_POWER
IRP_MN_WAIT_WAKE

For all other power requests, Mouclass copies the current IRP stack location, requests the next power request, and sends the request down the device stack.

For more information on the generic operation of these requests, see *I/O Request for Power Management* in Volume 1 of the *Windows 2000 Driver Development Reference*.

Status I/O Block

Under normal operation the status block values are specific to the minor function request. If a power request is sent in error to a grandmaster device or a legacy device, Mouclass completes the request with a status of STATUS_NOT_SUPPORTED.

IRP_MJ_READ

Operation

The IRP_MJ_READ request transfers zero or more MOUSE_INPUT_DATA structures from the internal data queue to the Win32 subsystem buffer. If there is no data in the data queue, a read request remains pending until it is completed or canceled.

The read request is not sent down the device stack and is completed by Mouclass.

A read request can be canceled. A read request is not executed if a cleanup is in progress when the request is received.

Note that a read request can be completed successfully only if it is made by a trusted subsystem. The Win32 subsystem is currently the only trusted subsystem. Mouclass performs a privilege check to enforce this restriction.

Input

Parameters.Read.Length member specifies the size in bytes of zero or more of MOUSE_INPUT_DATA structures:

```
typedef struct MOUSE_INPUT_DATA {
    USHORT UnitId;      // zero-based unit number of the mouse port
    USHORT Flags;       // indicator flags
    union {
        ULONG Buttons; // transition state of the mouse buttons
        struct {
            USHORT ButtonFlags; // transition state of mouse buttons
            USHORT ButtonData;  // data for flags (such as amount
                                // of movement if MOUSE_WHEEL is set)
        };
    };
    ULONG RawButtons;  // the raw state of the mouse buttons,
                       // currently not used by the Win32 subsystem
    LONG LastX;        // the signed relative or absolute motion
                       // in the X direction
    LONG LastY;        // the signed relative or absolute motion
                       // in the Y direction
    ULONG ExtraInformation;  // device-specific information
                             // for the event
} MOUSE_INPUT_DATA, *PMOUSE_INPUT_DATA
```

Output

The **AssociatedIrp.SystemBuffer** member points to an output buffer that is allocated by the Win32 subsystem to output the requested number of MOUSE_INPUT_DATA structures.

I/O Status Block

The **Information** member specifies the number of bytes transferred to the Win32 subsystem buffer. The number of bytes that are transferred is the smallest of the requested number of bytes and the number of bytes currently in the data queue.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

At least one MOUSE_INPUT_DATA structure was transferred.

STATUS_BUFFER_TOO_SMALL

The number of requested bytes is not an integer multiple of the size in bytes of a MOUSE_INPUT_DATA structure.

STATUS_PRIVILEGE_NOT_HELD

The requesting subsystem does not have read privileges.

STATUS_CANCELLED

The request was canceled before the transfer actually took place.

IRP_MJ_SYSTEM_CONTROL

Operation

Mouclass supports the following Windows Management Instrumentation (WMI) minor functions:

IRP_MN_REGINFO
IRP_MN_QUERY_DATA_BLOCK
IRP_MN_CHANGE_SINGLE_INSTANCE
IRP_MN_CHANGE_SINGLE_ITEM

For all other system control requests, Mouclass skips the current IRP and sends the request down the device stack to be completed by a lower-level driver.

Mouclass calls **WmiSystemControl** to process WMI requests. Mouclass registers two types of WMI data blocks: one type holds a pointer to a class device object, the other type indicates whether or not the class device supports a *wait/wake* operation. Wait/wake operation can be enabled or disabled. If wait/wake operation is supported and enabled, the device wakes the system after a wake event occurs. If wait/wake operation is not supported or disabled, the device can not wake the system.

When Called

Mouclass receives an IRP_MN_REGINFO request after it calls **IoWMIRegistrationControl** to update registration information. Mouclass updates registration information when a device is started or removed.

At the request of a WMI client, the WDM provider sends one of the following requests to Mouclass:

- An IRP_MN_QUERY_DATA_BLOCK request to obtain the data in one of registered data blocks.
- An IRP_MN_CHANGE_SINGLE_*Xxx* request to change the data in one of registered data blocks.

I/O Status Block

If the request is handled by the driver, the **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_DEVICE_REQUEST

STATUS_WMI_GUID_NOT_FOUND

The requested data block is not valid.

STATUS_WMI_INSTANCE_NOT_FOUND

The WMI context is not valid.

Mouclass Device Control Requests

This section describes the following device control request that Mouclass supports:

IOCTL_MOUSE_QUERY_ATTRIBUTES

Mouclass changes this device control request into an internal device control request and sends the changed request down the device stack.

Mouclass also sends a number of other device control requests for Plug and Play devices down the device stack without changing the request to an internal device control request. For a list of these requests, see *IOCTL_Xxx Device Control Requests.*

Filter drivers between the class and function drivers can filter these requests before they send the request down the device stack or after the lower-level drivers complete the request.

If Mouclass does not support the request, it completes the request with a status of STATUS_INVALID_DEVICE_REQUEST.

IOCTL_MOUSE_QUERY_ATTRIBUTES

Operation

The IOCTL_MOUSE_QUERY_ATTRIBUTES request returns information about the mouse attributes.

Mouclass copies the current stack location, sets the **MajorFunction** member of the new stack location to IRP_MJ_INTERNAL_DEVICE_CONTROL, and sends this request down the device stack.

For more information on this request, see *I8042prt Mouse Internal Device Control Requests* in Chapter 8.

Input

The **Parameters.DeviceIoControl.InputBufferLength** member is set to zero or a value greater than or equal to the size in bytes of a MOUSE_UNIT_ID_PARAMETER. A value of zero specifies a default unit ID of zero.

The **AssociatedIrp.SystemBuffer** member points to a client-allocated buffer that is used to input and output information. On input, **AssociatedIrp.SystemBuffer** points to a MOUSE_UNIT_ID_PARAMETER structure. The client sets the **UnitId** member of the input structure.

The **Parameters.DeviceIoControl.OutputBufferLength** member specifies the size in bytes of an output buffer, which must be greater than or equal to the size in bytes of a MOUSE_ATTRIBUTES structure.

Output

AssociatedIrp.SystemBuffer points to the client-allocated buffer that the lower-level drivers use to output a MOUSE_ATTRIBUTES structure.

I/O Status Block

The **Information** member is set to the number of bytes of attribute data that are returned if the request is successful.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

The **UnitId** value is invalid.

STATUS_BUFFER_TOO_SMALL

The **Parameters.DeviceIoControl.InputBufferLength** value is greater than zero but less than the size in bytes of a MOUSE_UNIT_ID_PARAMETER structure.

STATUS_NOT_SUPPORTED

The target device is associated with a subordinate class device.

IOCTL_Xxx Device Control Requests

Mouclass skips the current IRP stack and sends the following control requests down the device stack to be completed by a lower-level driver:

IOCTL_GET_SYS_BUTTON_CAPS
IOCTL_GET_SYS_BUTTON_EVENT
IOCTL_HID_GET_DRIVER_CONFIG
IOCTL_HID_SET_DRIVER_CONFIG
IOCTL_HID_GET_POLL_FREQUENCY_MSEC
IOCTL_HID_SET_POLL_FREQUENCY_MSEC
IOCTL_GET_NUM_DEVICE_INPUT_BUFFERS
IOCTL_SET_NUM_DEVICE_INPUT_BUFFERS
IOCTL_HID_GET_COLLECTION_INFORMATION
IOCTL_HID_GET_COLLECTION_DESCRIPTOR
IOCTL_HID_FLUSH_QUEUE
IOCTL_HID_SET_FEATURE
IOCTL_HID_GET_FEATURE
IOCTL_GET_PHYSICAL_DESCRIPTOR

IOCTL_HID_GET_HARDWARE_ID
IOCTL_HID_GET_MANUFACTURER_STRING
IOCTL_HID_GET_PRODUCT_STRING
IOCTL_HID_GET_SERIALNUMBER_STRING
IOCTL_HID_GET_INDEXED_STRING

Mouclass Class Service Callback Routine

This section describes **MouseClassServiceCallback**, the mouse class service callback routine.

Mouclass uses an IOCTL_INTERNAL_MOUSE_CONNECT request to connect its class service callback to a mouse device.

MouseClassServiceCallback

```
VOID
  MouseClassServiceCallback (
    IN PDEVICE_OBJECT DeviceObject,
    IN PMOUSE_INPUT_DATA InputDataStart,
    IN PMOUSE_INPUT_DATA InputDataEnd,
    IN OUT PULONG InputDataConsumed
    );
```

The **MouseClassServiceCallback** routine is the class service callback that is provided by Mouclass. A function driver calls the class service callback in its ISR dispatch completion routine. The class service callback transfers input data from the input data buffer of a device to the class data queue.

Parameters

DeviceObject

Pointer to the class device object.

InputDataStart

Pointer to the first mouse data packet, a MOUSE_INPUT_DATA structure, in the port device's input buffer.

InputDataEnd

Pointer to the mouse data packet that immediately follows the last data packet in the port device's input data buffer.

InputDataConsumed

Pointer to the number of mouse data packets that are transferred by the routine.

Include

mouclass.h

Comments

MouseClassServiceCallback transfers input data from the input buffer of the device to the class data queue. This routine is called by the ISR dispatch completion routine of the function driver.

MouseClassServiceCallback can be supplemented by a filter service callback that is provided by an upper-level mouse filter driver. A filter service callback can filter the mouse data that is transferred to the class data queue. For example, the filter service callback can delete, transform, or insert data. *Moufiltr*, the sample filter driver in the Microsoft® Windows® 2000 DDK, includes **MouFilter_ServiceCallback**, which is a template for a filter service callback.

MouseClassServiceCallback runs in kernel mode at IRQL DISPATCH_LEVEL.

CHAPTER 8

I8042prt Driver Reference

This chapter includes the following topics about *I8042prt*, the Microsoft® Windows® 2000 system function driver for a PS/2-style keyboard device and a PS/2-style mouse device:

- *I8042prt Keyboard Major I/O Requests*
- *I8042prt Keyboard Internal Device Control Requests*
- *I8042prt Mouse Major I/O Requests*
- *I8042prt Mouse Internal Device Control Requests*
- *I8042prt Keyboard Callback Routines*
- *I8042prt Mouse Callback Routines*

Note that the operational constraints of I8042prt do not apply to *Sermouse*, the Windows 2000 system function driver for a serial mouse.

For more information on I8042prt operation, see the following topics:

- *Keyboard and Mouse Drivers for Non-HID Devices* in the online DDK
- Include files *ntddkbd.h* and *ntddmou.h* in the *%user's install path%\inc* directory in the Windows 2000 DDK
- Sample code in the *%user's install path%\src\input* directory in the Windows 2000 DDK

I8042prt Keyboard Major I/O Requests

This section describes the I8042prt-specific operation of the following major I/O requests that I8042prt supports for a keyboard device:

IRP_MJ_CLOSE
IRP_MJ_CREATE
IRP_MJ_PNP

IRP_MJ_POWER
IRP_MJ_SYSTEM_CONTROL

For information on how I8042prt handles the generic operation of these requests, see *IRP Function Codes and IOCTLs* in Part 1.

IRP_MJ_CREATE

Operation

The IRP_MJ_CREATE request opens a file on a keyboard device.

Note that Kbdclass uses an IOCTL_INTERNAL_KEYBOARD_CONNECT request to connect to a device before Kbdclass can open the device. There can be only one connection to the device. For more information, see *Open and Close a Keyboard and Mouse Device* in Part 4 in the *Kernel-Mode Drivers Design Guide*.

I8042prt completes the IRP_MJ_CREATE request.

I/O Status Block

The **Information** member is set to zero.

The **Status** member is set to one of the following status codes:

STATUS_SUCCESS

STATUS_INVALID_DEVICE_STATE

The keyboard class service is not connected.

STATUS_NO_SUCH_DEVICE

The keyboard device is not present.

IRP_MJ_DEVICE_CONTROL

I8042prt supports the following device control requests for a keyboard device:

IOCTL_GET_SYS_BUTTON_CAPS
IOCTL_GET_SYS_BUTTON_EVENT

These requests are used exclusively by the Power Manager. No other device control requests are supported.

If the device is in a Plug and Play stopped state or removed state, or a request is not supported, I8042prt completes the request with a status of STATUS_INVALID_DEVICE_REQUEST.

IRP_MJ_INTERNAL_DEVICE_CONTROL

I8042prt supports the following internal device control requests:

IOCTL_INTERNAL_I8042_CONTROLLER_WRITE_BUFFER
IOCTL_INTERNAL_I8042_HOOK_KEYBOARD
IOCTL_INTERNAL_I8042_KEYBOARD_START_INFORMATION
IOCTL_INTERNAL_I8042_KEYBOARD_WRITE_BUFFER
IOCTL_INTERNAL_KEYBOARD_CONNECT
IOCTL_INTERNAL_KEYBOARD_DISCONNECT
IOCTL_KEYBOARD_QUERY_ATTRIBUTES
IOCTL_KEYBOARD_QUERY_INDICATOR_TRANSLATION
IOCTL_KEYBOARD_QUERY_INDICATORS
IOCTL_KEYBOARD_QUERY_TYPEMATIC
IOCTL_KEYBOARD_SET_INDICATORS
IOCTL_KEYBOARD_SET_TYPEMATIC

For more information on these requests, see *I8042prt Keyboard Internal Device Control Requests*.

I8042prt completes all other requests with a status of STATUS_INVALID_DEVICE_REQUEST.

IRP_MJ_PNP

Operation

I8042prt processes the following Plug and Play requests for a keyboard device:

IRP_MN_CANCEL_REMOVE_DEVICE
IRP_MN_CANCEL_STOP_DEVICE
IRP_MN_FILTER_RESOURCE_REQUIREMENTS
IRP_MN_QUERY_PNP_DEVICE_STATE
IRP_MN_QUERY_REMOVE_DEVICE
IRP_MN_QUERY_STOP_DEVICE
IRP_MN_REMOVE_DEVICE
IRP_MN_START_DEVICE

For all other Plug and Play requests, I8042prt skips the current IRP stack and sends the request down the device stack.

For more information on the generic operation of these requests, see *Plug and Play IRPs*.

I/O Status Block

The status block values are function-specific.

IRP_MJ_POWER

Operation

I8042prt processes the following power request for a keyboard device:

IRP_MN_SET_POWER

For all other power requests, I8042prt skips the current IRP stack, requests the next power IRP, and sends the request down the device stack.

For more information on the generic operation of these requests, see *I/O Requests for Power Management* in Volume 1 of the *Windows 2000 Driver Development Reference*.

Status I/O Block

The status block values are function-specific.

IRP_MJ_SYSTEM_CONTROL

Operation

I8042prt supports the following Windows® Management Instrumentation (WMI) system control requests for a keyboard device:

IRP_MN_REGINFO
IRP_MN_QUERY_DATA_BLOCK

For all other system control requests, I8042prt skips the current IRP stack location, and sends the request down the device stack to be completed by a lower-level driver.

I8042prt calls **WmiSystemControl** to process WMI requests. I8042prt registers the WMI data block that contains a KEYBOARD_PORT_WMI_STD_DATA structure.

When Called

The WDM provider sends an IRP_MN_REGINFO request after a driver calls **IoWMI-RegistrationControl** to update WMI registration information—see ***DpWmiQueryReginfo* in Part 8**. The port driver updates WMI registration information when the device is started and removed.

At the request of a WMI client, the WDM provider sends an IRP_MN_QUERY_DATA_BLOCK request to the port driver to obtain a data block containing a keyboard KEYBOARD_PORT_WMI_STD_DATA structure.

I/O Status Block

If the request is handled by I8042prt, the **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_DEVICE_REQUEST

STATUS_WMI_GUID_NOT_FOUND

The data block GUID is not valid.

STATUS_WMI_INSTANCE_NOT_FOUND

The WMI context is not valid.

I8042prt Keyboard Internal Device Control Requests

This section describes the following internal device control requests that I8042prt supports for a keyboard device:

IOCTL_INTERNAL_I8042_CONTROLLER_WRITE_BUFFER
IOCTL_INTERNAL_I8042_HOOK_KEYBOARD
IOCTL_INTERNAL_I8042_KEYBOARD_START_INFORMATION
IOCTL_INTERNAL_I8042_KEYBOARD_WRITE_BUFFER
IOCTL_INTERNAL_KEYBOARD_CONNECT
IOCTL_INTERNAL_KEYBOARD_DISCONNECT
IOCTL_KEYBOARD_QUERY_ATTRIBUTES
IOCTL_KEYBOARD_QUERY_INDICATOR_TRANSLATION
IOCTL_KEYBOARD_QUERY_INDICATORS
IOCTL_KEYBOARD_QUERY_TYPEMATIC
IOCTL_KEYBOARD_SET_INDICATORS
IOCTL_KEYBOARD_SET_TYPEMATIC

I8042prt completes all other internal device control with a status of STATUS_INVALID_DEVICE_REQUEST.

IOCTL_INTERNAL_I8042_CONTROLLER_WRITE_BUFFER

Operation

The IOCTL_INTERNAL_I8042_CONTROLLER_WRITE_BUFFER request is not supported.

Status I/O Block

The **Status** member is set to STATUS_NOT_SUPPORTED.

IOCTL_INTERNAL_I8042_HOOK_KEYBOARD

Operation

The IOCTL_INTERNAL_I8042_HOOK_KEYBOARD request adds the following callback routines to I8042prt's operation:

- An optional initialization callback routine that I8042prt calls when it initializes a keyboard
- An optional callback routine into I8042prt's interrupt service routine

These optional callback routines are added by an upper-level filter driver for the keyboard device.

After I8042prt receives an IOCTL_INTERNAL_KEYBOARD_CONNECT request, it sends a synchronous IOCTL_INTERNAL_I8042_HOOK_KEYBOARD request to the top of the keyboard device stack. When the upper-level filter driver receives this request, the filter driver sets the keyboard **IsrRoutine** member and the keyboard **InitializationRoutine** member of the INTERNAL_I8042_HOOK_KEYBOARD structure passed with the request.

Input

The **Parameters.DeviceIoControl.Type3InputBuffer** points to an INTERNAL_I8042_HOOK_KEYBOARD structure. This structure includes:

- The **InitializationRoutine** member that points to a callback routine that is called by I8042prt's initialization service routine
- The **IsrRoutine** member that points to a callback routine that is called by I8042prt's interrupt service routine

I/O Status Block

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.Type3InputBuffer is less than the size in bytes of an INTERNAL_I8042_HOOK_KEYBOARD structure.

IOCTL_INTERNAL_I8042_KEYBOARD_START_INFORMATION

Operation

The IOCTL_INTERNAL_I8042_KEYBOARD_START_INFORMATION request passes a pointer to a keyboard interrupt object. I8042prt sends this request synchronously to the top of the device stack after the keyboard interrupt object is created. Upper-level filter drivers that need to synchronize their callback operation with the keyboard interrupt service routine (ISR) can use the pointer to the keyboard interrupt object. For more information on this request, see *Synchronize the Operation of a Filter Driver with a Device's Interrupt Service Routine* in Part 4 in the *Kernel-Mode Drivers Design Guide*.

Input

The **AssociatedIrp.SystemBuffer** points to a buffer allocated by I8042prt to input an INTERNAL_I8042_START_INFORMATION structure.

The **Parameters.DeviceIoControl.InputBufferLength** specifies the size in bytes of an INTERNAL_I8042_START_INFORMATION structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to STATUS_SUCCESS.

IOCTL_INTERNAL_I8042_KEYBOARD_WRITE_BUFFER

Operation

The IOCTL_INTERNAL_I8042_KEYBOARD_WRITE_BUFFER request writes data to the i8042 controller to control operation of a keyboard device. A filter driver can use this request to control the operation of a keyboard.

I8042prt synchronizes write buffer requests and other keyboard requests that write to the i8042 controller, including IOCTL_KEYBOARD_SET_INDICATORS and IOCTL_KEYBOARD_SET_TYPEMATIC. I8042prt synchronizes the actual write of data with the keyboard ISR.

Input

The **Parameters.DeviceIoControl.InputBufferLength** is set to the number of bytes in the input buffer, which must be greater than one.

The **Parameters.DeviceIoControl.Type3InputBuffer** points to a client-allocated buffer to input the data to write to an i8042 controller.

Status I/O Block

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_DEVICE_NOT_READY

The keyboard interrupt is not initialized.

STATUS_INVALID_PARAMETER

The input parameters are not valid.

STATUS_IO_TIMEOUT

The request timed out.

IOCTL_INTERNAL_KEYBOARD_CONNECT

Operation

The IOCTL_INTERNAL_KEYBOARD_CONNECT request connects the Kbdclass service to the keyboard device.

After I8042prt receives a keyboard connect request, it sends a synchronous IOCTL_INTERNAL_I8042_HOOK_KEYBOARD request to the top of keyboard device stack. The

connect request is completed after the IOCTL_INTERNAL_I8042_HOOK_KEYBOARD request is completed.

For more information, see *Connect a Class Service Callback and a Filter Service Callback to a Device* in the online DDK.

Input

The **Parameters.DeviceIoControl.Type3InputBuffer** points to a CONNECT_DATA structure. This structure includes a pointer to a device object and a pointer to a class service callback routine. The connect data is set by Kbdclass and can be filtered (reset) by a filter driver below Kbdclass in the device stack.

The **Parameters.DeviceIoControl.InputBufferLength** specifies the size in bytes of a CONNECT_DATA structure.

I/O Status Block

The **Information** member is set by the port driver.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of a CONNECT_DATA structure.

STATUS_SHARING_VIOLATION

The port driver is already connected.

IOCTL_INTERNAL_KEYBOARD_DISCONNECT

Operation

The IOCTL_INTERNAL_KEYBOARD_DISCONNECT request is not implemented.

Note that the Plug and Play Manager can dynamically add and remove Plug and Play devices.

I/O Status Block

The **Status** member is set to STATUS_NOT_IMPLEMENTED.

IOCTL_KEYBOARD_QUERY_ATTRIBUTES

Operation

The IOCTL_KEYBOARD_QUERY_ATTRIBUTES request returns information about the keyboard attributes.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to a value greater than or equal to the size in bytes of a KEYBOARD_ATTRIBUTES structure.

Output

AssociatedIrp.SystemBuffer points to a client-allocated buffer that I8042prt uses to output a KEYBOARD_ATTRIBUTES structure.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of a KEYBOARD_ATTRIBUTES structure. Otherwise the **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

Parameters.DeviceIoControl.OutputBufferLength is less than the size in bytes of a KEYBOARD_ATTRIBUTES structure.

IOCTL_KEYBOARD_QUERY_INDICATORS

Operation

The IOCTL_KEYBOARD_QUERY_INDICATORS request returns information about the keyboard indicators.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to a value greater than or equal to the size in bytes of a KEYBOARD_INDICATOR_PARAMETERS structure.

Output

AssociatedIrp.SystemBuffer points to a client-allocated buffer that I8042prt uses to output a KEYBOARD_INDICATOR_PARAMETERS structure.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of a KEYBOARD_INDICATOR_PARAMETERS structure. Otherwise, **Information** is set to zero.

The **Status** member is set to one the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

Parameters.DeviceIoControl.OutputBufferLength is less than the size in bytes of a KEYBOARD_INDICATOR_PARAMETERS structure.

IOCTL_KEYBOARD_QUERY_INDICATOR_TRANSLATION

Operation

The IOCTL_KEYBOARD_QUERY_INDICATOR_TRANSLATION request returns information about the mapping between scan codes and keyboard indicators.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** is set to a value greater than or equal to the size in bytes of a device-specific KEYBOARD_INDICATOR_TRANSLATION structure. This structure includes a variable-sized array of INDICATOR_LIST members that is device-specific.

Output

AssociatedIrp.SystemBuffer points to a client-allocated buffer that I8042prt uses to output a KEYBOARD_INDICATOR_TRANSLATION structure. This structure includes a variable-sized array of INDICATOR_LIST members that is device-specific.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of the device-specific KEYBOARD_INDICATOR_TRANSLATION structure. Otherwise, **Information** is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

Parameters.DeviceIoControl.OutputBufferLength is less than the size in bytes of the device-specific KEYBOARD_INDICATORS_TRANSLATION structure.

IOCTL_KEYBOARD_QUERY_TYPEMATIC

Operation

The IOCTL_KEYBOARD_QUERY_TYPEMATIC request returns the keyboard typematic settings.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to a value greater than or equal to the size in bytes of a KEYBOARD_TYPEMATIC_PARAMETERS structure.

Output

AssociatedIrp.SystemBuffer points to a client-allocated output buffer that I8042prt uses to output a KEYBOARD_TYPEMATIC_PARAMETERS structure.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of a KEYBOARD_TYPEMATIC_PARAMETERS structure. Otherwise, **Information** is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

Parameters.DeviceIoControl.OutputBufferLength is less than the size in bytes of a KEYBOARD_TYPEMATIC_PARAMETERS structure.

IOCTL_KEYBOARD_SET_INDICATORS

Operation

The IOCTL_KEYBOARD_SET_INDICATORS request sets the keyboard indicators.

Input

The **AssociatedIrp.SystemBuffer** member points to a client-allocated buffer to input a KEYBOARD_INDICATORS_PARAMETERS structure. The client sets the indicator parameters in this structure.

The **Parameters.DeviceIoControl.InputBufferLength** member is set to a value greater than or equal to the size in bytes of a KEYBOARD_INDICATORS_PARAMETERS structure.

I/O Status Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_DEVICE_NOT_READY

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of a KEYBOARD_INDICATORS_PARAMETERS structure, or the specified indicator parameters are invalid.

STATUS_IO_TIMEOUT

The request timed out.

IOCTL_KEYBOARD_SET_TYPEMATIC

Operation

The IOCTL_KEYBOARD_SET_TYPEMATIC request sets the keyboard typematic settings.

Input

The **AssociatedIrp.SystemBuffer** member points to a client-allocated buffer to input a KEYBOARD_TYPEMATIC_PARAMETERS structure. The client sets the typematic parameters in this structure.

The **Parameters.DeviceIoControl.InputBufferLength** member is set to a value greater than or equal to the size in bytes of a KEYBOARD_TYPEMATIC_PARAMETERS structure.

I/O Status Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_DEVICE_NOT_READY

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of a KEYBOARD_TYPEMATIC_PARAMETERS structure, or the specified typematic settings are invalid.

STATUS_IO_TIMEOUT

The request timed out.

I8042prt Mouse Major I/O Requests

This section describes the I8042prt-specific operation of the following major I/O requests for a mouse device:

IRP_MJ_CLOSE
IRP_MJ_CREATE
IRP_MJ_DEVICE_CONTROL
IRP_MJ_INTERNAL_DEVICE_CONTROL
IRP_MJ_PNP
IRP_MJ_POWER
IRP_MJ_SYSTEM_CONTROL

For the generic operation of these requests, see *IRP Function Codes and IOCTLs*.

IRP_MJ_CREATE

Operation

The IRP_MJ_CREATE request opens a file on a mouse device.

Note that Mouclass uses an IOCTL_INTERNAL_MOUSE_CONNECT request to connect to a mouse device before Mouclass can open the device. There can be only one connection to the device. For more information, see *Open and Close a Keyboard and Mouse Device*, in Part 4 in the *Kernel-Mode Drivers Design Guide*.

I8042prt completes the IRP_MJ_CREATE request.

I/O Status Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_DEVICE_STATE

The Mouclass service is not connected.

STATUS_NO_SUCH_DEVICE

The device is not present.

IRP_MJ_DEVICE_CONTROL

I8042prt does not support any device control requests for a mouse device.

I8042prt completes all device control requests with a status of STATUS_INVALID_DEVICE_REQUEST.

IRP_MJ_INTERNAL_DEVICE_CONTROL

I8042prt supports the following internal device control requests for a mouse device:

IOCTL_INTERNAL_I8042_HOOK_MOUSE
IOCTL_INTERNAL_I8042_MOUSE_START_INFORMATION
IOCTL_INTERNAL_I8042_MOUSE_WRITE_BUFFER
IOCTL_INTERNAL_MOUSE_CONNECT
IOCTL_INTERNAL_MOUSE_DISCONNECT
IOCTL_MOUSE_QUERY_ATTRIBUTES

I8042prt completes all other internal device control requests with a status of STATUS_INVALID_DEVICE_REQUEST.

IRP_MJ_PNP

I8042prt processes the following Plug and Play requests for a mouse device:

IRP_MN_CANCEL_REMOVE_DEVICE
IRP_MN_CANCEL_STOP_DEVICE
IRP_MN_FILTER_RESOURCE_REQUIREMENTS
IRP_MN_QUERY_PNP_DEVICE_STATE
IRP_MN_QUERY_REMOVE_DEVICE
IRP_MN_QUERY_STOP_DEVICE
IRP_MN_REMOVE_DEVICE
IRP_MN_START_DEVICE

For all other Plug and Play requests, I8042prt skips the current IRP stack and sends the request down the device stack.

For more information on the generic operation of these requests, see *Plug and Play IRPs* in Volume 1 of *Windows 2000 Driver Development Reference*.

I/O Status Block

The values of the status block members are function-specific.

IRP_MJ_POWER

Operation

I8042prt processes the following power request for a mouse device:

IRP_MN_SET_POWER

For all other power requests, I8042prt skips the current IRP stack, requests the next power IRP, and sends the request down the device stack.

For more information on the generic operation of these requests, see *I/O Requests for Power Management* in Volume 1 of *Windows 2000 Driver Development Reference*.

Status I/O Block

The values of the status block members are function-specific.

IRP_MJ_SYSTEM_CONTROL

Operation

I8042prt supports the following Windows® Management Instrumentation (WMI) system control requests for a mouse device:

IRP_MN_REGINFO
IRP_MN_QUERY_DATA_BLOCK

For all other system control requests, I8042prt skips the current IRP stack location and sends the request down the device stack to be completed by a lower-level driver.

I8042prt calls **WmiSystemControl** to process WMI requests. I8042prt registers a WMI data block that contains a POINTER_PORT_WMI_STD_DATA structure.

When Called

The WDM provider sends an IRP_MN_REGINFO request after the port driver calls **IoWMIRegistrationControl** to update WMI registration information—see ***DpWmiQueryReginfo*** in Part 8. I8042prt updates WMI registration information when the device is started and removed.

At the request of a WMI client, the WDM provider sends an IRP_MN_QUERY_DATA_BLOCK request to I8042prt. This request obtains a WMI data block containing a POINTER_PORT_WMI_STD_DATA structure.

I/O Status Block

If I8042prt handles the request, the **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_DEVICE_REQUEST

STATUS_WMI_GUID_NOT_FOUND
The data block GUID is not valid.

STATUS_WMI_INSTANCE_NOT_FOUND
The WMI context is not valid.

I8042prt Mouse Internal Device Control Requests

I8042prt supports the following internal device control requests for a mouse device:

IOCTL_INTERNAL_I8042_HOOK_MOUSE
IOCTL_INTERNAL_I8042_MOUSE_START_INFORMATION
IOCTL_INTERNAL_I8042_MOUSE_WRITE_BUFFER
IOCTL_INTERNAL_MOUSE_CONNECT
IOCTL_INTERNAL_MOUSE_DISCONNECT
IOCTL_MOUSE_QUERY_ATTRIBUTES

I8042prt completes all other device control requests with a status of STATUS_INVALID_DEVICE_REQUEST.

IOCTL_INTERNAL_I8042_HOOK_MOUSE

Operation

The IOCTL_INTERNAL_I8042_HOOK_MOUSE request adds an optional callback routine into the interrupt service routine that I8042prt provides for a mouse device.

The optional callback routine is provided by an optional upper-level filter driver for the mouse device.

After I8042port receives an *IOCTL_INTERNAL_MOUSE_CONNECT request* request, it sends a synchronous IOCTL_INTERNAL_I8042_HOOK_MOUSE request to the top of the keyboard device stack. When the upper-level filter driver receives this request, the filter driver sets the mouse **IsrRoutine** member of the INTERNAL_I8042_HOOK_MOUSE structure that is passed with the request.

Input

The **Parameters.DeviceIoControl.Type3InputBuffer** points to an INTERNAL_I8042_ HOOK_MOUSE structure. This structure includes an **IsrRoutine** member that points to a callback that is called by the interrupt service routine that I8042prt provides for a mouse device.

I/O Status Block

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.Type3InputBuffer is less than the size in bytes of an INTERNAL_I8042_HOOK_MOUSE structure.

IOCTL_INTERNAL_I8042_MOUSE_START_INFORMATION

Operation

The IOCTL_INTERNAL_I8042_MOUSE_START_INFORMATION request passes a pointer to a mouse interrupt object. I8042prt sends this request synchronously to the top of the device stack after the mouse interrupt object is created. Upper-level filter drivers that need to synchronize their callback operation with the mouse ISR can use the pointer to the mouse interrupt object. For more information on this request, see *Synchronize the Operation of a Filter Driver with a Device's ISR* in Part 4 in the *Kernel-Mode Drivers Design Guide*.

Input

The **AssociatedIrp.SystemBuffer** points to an input buffer allocated by I8042prt to input an INTERNAL_I8042_START_INFORMATION structure.

The **Parameters.DeviceIoControl.InputBufferLength** specifies the size in bytes of an INTERNAL_I8042_START_INFORMATION structure.

Status I/O Block

The **Information** member is set to zero.

The **Status** member is set to STATUS_SUCCESS.

IOCTL_INTERNAL_I8042_MOUSE_WRITE_BUFFER

Operation

The IOCTL_INTERNAL_I8042_MOUSE_WRITE_BUFFER request writes data to the i8042 controller to control operation of a mouse device. An upper-level filter driver can use this request to control the operation of a mouse.

I8042prt synchronizes write buffer requests with one another. I8042prt synchronizes the actual write of data with the mouse ISR.

Input

The **Parameters.DeviceIoControl.InputBufferLength** is set to the number of bytes in the input buffer, which must be greater than one.

The **Parameters.DeviceIoControl.Type3InputBuffer** points to a client-allocated buffer to input the data to write to an i8042 controller.

Status I/O Block

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_DEVICE_NOT_READY

The mouse interrupt is not initialized.

STATUS_INVALID_PARAMETER

The input parameters are not valid.

STATUS_IO_TIMEOUT

The request timed out.

IOCTL_INTERNAL_MOUSE_CONNECT

Operation

The IOCTL_INTERNAL_MOUSE_CONNECT request connects the Mouclass service to a mouse device.

After I8042prt receives a mouse connect request, it sends a synchronous INTERNAL_I8042_HOOK_MOUSE request to the top of mouse device stack. The connect request is completed after the INTERNAL_I8042_HOOK_MOUSE request is completed.

For more information, see *Connect a Class Service Callback and a Filter Service Callback to a Device* in Part 4 in the *Kernel-Mode Drivers Design Guide*.

Input

The **Parameters.DeviceIoControl.Type3InputBuffer** points to a CONNECT_DATA structure. This structure includes a pointer to a device object and a pointer to a mouse class service callback routine. The connect data is set by the class service and can be filtered (reset) by filter drivers.

The **Parameters.DeviceIoControl.InputBufferLength** specifies the size in bytes of a CONNECT_DATA structure.

I/O Status Block

The **Information** member is set by the port driver.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of a CONNECT_DATA structure.

STATUS_SHARING_VIOLATION

The port driver is already connected.

IOCTL_INTERNAL_MOUSE_DISCONNECT

Operation

The IOCTL_INTERNAL_MOUSE_DISCONNECT request is not implemented.

Note that the Plug and Play Manager can dynamically add and remove Plug and Play devices.

I/O Status Block

The **Status** member is set to STATUS_NOT_IMPLEMENTED.

IOCTL_MOUSE_QUERY_ATTRIBUTES

Operation

The IOCTL_MOUSE_QUERY_ATTRIBUTES request returns information about the mouse attributes.

Input

The **Parameters.DeviceIoControl.OutputBufferLength** member is set to a value greater than or equal to the size in bytes of a MOUSE_ATTRIBUTES structure.

Output

AssociatedIrp.SystemBuffer points to a client-allocated output buffer that I8042prt uses to output a MOUSE_ATTRIBUTES structure.

I/O Status Block

If the request is successful, the **Information** member is set to the size in bytes of a MOUSE_ATTRIBUTES structure. Otherwise, the **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_BUFFER_TOO_SMALL

Parameters.DeviceIoControl.OutputBufferLength is less than the size in bytes of a MOUSE_ATTRIBUTES structure.

I8042prt Keyboard Callback Routines

This section describes the following keyboard callback routines that I8042prt supports:

- Callbacks that an upper-level keyboard filter driver can provide to supplement the operation of I8042prt:
  - *Keyboard* ***InitializationRoutine***
  - *Keyboard* ***IsrRoutine***
- Callbacks that I8042prt provides that an upper-level keyboard filter driver can use:
  - *Keyboard* ***IsrWritePort***
  - ***QueueKeyboardInput***

I8042prt uses an IOCTL_INTERNAL I8042_HOOK_KEYBOARD request to add, or *hook*, the filter driver callbacks into the operation of I8042prt. The hook request also passes pointers to the I8042prt keyboard callbacks to the filter driver.

Keyboard InitializationRoutine

```
NTSTATUS
  (*InitializationRoutine)(
    IN PVOID InitializationContext,
    IN PVOID SynchFuncContext,
    IN PI8042_SYNCH_READ_PORT ReadPort,
    IN PI8042_SYNCH_WRITE_PORT WritePort,
    OUT PBOOLEAN TurnTranslationOn
    );
```

The keyboard **InitializationRoutine** callback routine supplements the default initialization of a keyboard device by I8042prt.

Parameters

InitializationContext

Pointer to the filter device object of the driver that supplies the callback routine.

SynchFuncContext

Pointer to the context for the routines pointed to by *ReadPort* and *Writeport.*

ReadPort

Pointer to a routine that reads from the port.

WritePort

Pointer to a routine that writes to the port.

TurnTranslationOn

Specifies whether to turn translation on or off. If *TranslationOn* is TRUE, translation is turned on; otherwise, translation is turned off.

Include

ntdd8042.h

Return Value

The keyboard **InitializationRoutine** callback returns an appropriate NTSTATUS code.

Comments

An upper-level keyboard filter driver can provide a keyboard **InitializationRoutine** callback routine.

If an upper-level keyboard filter driver supplies an initialization callback, I8042prt calls the filter initialization callback when I8042prt initializes the keyboard. Default keyboard initialization includes the following operations: reset the keyboard, set the typematic rate and delay, and set the light-emitting diodes (LED).

The keyboard **InitializationRoutine** callback runs in kernel mode at IRQL DISPATCH_LEVEL.

See Also

KbFilter_InitializationRoutine

Keyboard IsrRoutine

```
(*IsrRoutine)(
  IN PVOID IsrContext,
  IN PKEYBOARD_INPUT_DATA CurrentInput,
  IN POUTPUT_PACKET CurrentOutput,
  IN OUT UCHAR StatusByte,
  IN PUCHAR DataByte,
  OUT PBOOLEAN ContinueProcessing,
  IN PKEYBOARD_SCAN_STATE ScanState
  );
```

The keyboard **IsrRoutine** callback routine customizes the operation of a keyboard ISR.

Parameters

IsrContext

Pointer to the filter device object of the driver that supplies a callback routine.

CurrentInput

Pointer to the input KEYBOARD_DATA_INPUT structure that is being constructed by the ISR.

CurrentOutput

Pointer to the array of bytes that is being written to the hardware device.

StatusByte

Specifies the status byte that is read from I/O port 60 when an interrupt occurs.

DataByte

Specifies the data byte that is read from I/O port 64 when an interrupt occurs.

ContinueProcessing

Specifies whether processing in the ISR will continue after this routine completes.

ScanState

Specifies the keyboard scan state.

Include

ntdd8042.h

Return Value

The keyboard **IsrRoutine** callback returns TRUE if the interrupt service routine should continue; otherwise it returns FALSE.

Comments

A keyboard **IsrRoutine** callback routine is not needed if the default operation of I8042prt's ISR is sufficient.

An upper-level keyboard filter driver can provide a keyboard **IsrRoutine** callback. I8042prt's ISR calls the callback after it validates the interrupt and reads the scan code.

The keyboard **IsrRoutine** callback runs in kernel mode at the IRQL of the I8042prt's keyboard ISR.

See Also

KbFilter_IsrHook

Keyboard IsrWritePort

```
VOID
(*IsrWritePort)(
  IN PVOID  Context,
  IN UCHAR  Value
  )
```

The keyboard **IsrWritePort** callback routine writes data to a keyboard device. I8042prt provides this routine.

Parameters

Context

Pointer to the function device object that represents a keyboard device.

Value

Specifies the data to write to a keyboard device.

Include

ntdd8042.h

Comments

The keyboard **IsrWritePort** callback routine should only be called by a *keyboard **IsrRoutine*** callback routine. I8042prt calls the keyboard **IsrRoutine** callback in its keyboard ISR.

I8042prt passes a pointer to the keyboard **IsrWritePort** callback routine in the **IsrWritePort** member of the INTERNAL_I8042_HOOK_KEYBOARD structure that I8042prt uses with an IOCTL_INTERNAL_I8042_HOOK_KEYBOARD request.

See Also

I8042prt Callbacks that Filter Drivers Can Use in the online DDK, INTERNAL_I8042_HOOK_KEYBOARD

QueueKeyboardInput

```
(*QueueKeyboardInput)(
  IN PVOID  Context
  );
```

The **QueueKeyboardInput** callback routine queues a keyboard input data packet for processing by the ISR DPC of the keyboard. I8042prt provides this routine.

Parameters

Context

Pointer to the function device object that represents a keyboard device.

Include

ntdd8042.h

Comments

The **QueueKeyboardInput** callback routine should only be called by a *keyboard* ***IsrRoutine*** callback routine. I8042prt calls the keyboard **IsrRoutine** callback in its keyboard ISR. I8042prt passes a pointer to the callback in the **QueueKeyboardInput** member of the INTERNAL_I8042_HOOK_KEYBOARD structure that I8042prt uses with an IOCTL_INTERNAL_I8042_HOOK_KEYBOARD request.

See Also

I8042prt Callbacks that Filter Drivers Can Use in the online DDK

I8042prt Mouse Callback Routines

This section describes the following mouse callback routines that I8042prt supports:

- Callbacks that an upper-level mouse filter driver can provide to supplement the operation of I8042prt:
  - *Mouse* ***IsrRoutine***

- Callbacks that I8042prt provides that an upper-level mouse filter driver can use:
  - *Mouse* ***IsrWritePort***
  - ***QueueMouseInput***

I8042prt uses an IOCTL_INTERNAL_I8042_HOOK_MOUSE request to add, or *hook*, the filter driver callbacks into the operation of I8042prt. The hook request also passes pointers to the I8042prt mouse callbacks to the filter driver.

Mouse IsrRoutine

```
BOOLEAN
  (*IsrRoutine)(
    IN PVOID IsrContext,
    IN PMOUSE_INPUT_DATA CurrentInput,
    IN POUTPUT_PACKET CurrentOutput,
    IN UCHAR StatusByte,
    IN OUT PUCHAR Byte,
    OUT PBOOLEAN ContinueProcessing,
    IN PMOUSE_STATE MouseState,
    IN PMOUSE_RESET_SUBSTATE ResetSubState
    )
```

The mouse **IsrRoutine** routine customizes the operation of the mouse ISR.

Parameters

IsrContext

Pointer to the filter device object of the driver that supplies this callback routine.

CurrentInput

Pointer to the input MOUSE_DATA_INPUT structure being constructed by the ISR.

CurrentOutput

Pointer to the array of bytes being written to the hardware device.

StatusByte

Specifies a status byte that is read from I/O port 60 when the interrupt occurs.

Byte

Specifies a data byte that is read from I/O port 64 when the interrupt occurs.

ContinueProcessing

Specifies a flag indicating whether processing in the ISR will continue after this routine completes.

MouseState

Specifies the mouse state.

ResetSubState

Specifies a mouse substate.

Include

ntdd8042.h

Return Value

The mouse **IsrRoutine** callback routine returns TRUE if the interrupt service routine should continue; otherwise it returns FALSE.

Comments

A mouse **IsrRoutine** callback routine is not needed if the default operation of I8042prt is sufficient.

An upper-level keyboard filter driver can provide a mouse **IsrRoutine** callback. After the I8042prt's mouse ISR validates the interrupt, it calls the **IsrRoutine** callback.

The mouse **IsrRoutine** routine runs in kernel mode at the IRQL of I8042prt's mouse ISR.

See Also

MouFilter_IsrHook

Mouse IsrWritePort

```
VOID
(*IsrWritePort)(
  IN PVOID   Context,
  IN UCHAR   Value
  )
```

The mouse **IsrWritePort** callback routine writes data to a mouse device. I8042prt provides this routine.

Parameters

Context

Pointer to the function device object that represents a mouse device.

Value

Specifies the data to write to a mouse device.

Include

i8042prt.h

Comments

The mouse **IsrWritePort** callback routine should only be called by a *mouse **IsrRoutine*** callback routine. I8042prt calls the mouse **IsrRoutine** callback in its mouse ISR. I8042prt passes a pointer to this callback in the **IsrWritePort** member of an INTERNAL_I8042_HOOK_MOUSE structure that I8042prt uses with an IOCTL_INTERNAL_I8042_HOOK_MOUSE request.

See Also

I8042prt Callbacks that Filter Drivers Can Use in the online DDK

QueueMouseInput

```
(*QueueMousePacket)(
  IN PVOID  Context
  );
```

The **QueueMouseInput** callback routine queues a mouse input data packet for processing by the device's ISR deferred procedure call (DPC). I8042prt provides this routine.

Parameters

Context

Pointer to the function device object that represents a mouse device.

Include

ntdd8042prt.h

Comments

The **QueueMouseInput** callback routine should only be called by a mouse **IsrRoutine** callback routine. I8042prt calls the mouse **IsrRoutine** callback in its mouse ISR. I8042prt passes a pointer to the callback in the **QueueMouseInput** member of an INTERNAL_I8042_HOOK_MOUSE structure that I8042prt uses with an IOCTL_INTERNAL_I8042_HOOK_MOUSE request.

See Also

I8042prt Callbacks that Filter Drivers Can Use in the online DDK

CHAPTER 9

Kbfiltr Driver Reference

This chapter includes the following topics about *Kbfiltr*, the sample upper-level keyboard filter driver in the Microsoft® Windows® 2000 DDK:

- *Kbfiltr Internal Device Control Requests*
- *Kbfiltr Callback Routines*

Kbfiltr is designed to be used with *Kbdclass*, the Windows 2000 system class driver for keyboard devices, and *I8042prt*, the Windows 2000 function driver for a PS/2-style keyboard. Kbfiltr demonstrates how to filter I/O requests and how to add callback routines that modify the operation of Kbdclass and I8042prt.

See the following topics for more information about Kbfiltr operation:

- *Keyboard and Mouse Drivers for Non-HID Devices* in the online DDK
- Include file *%user's install path%\inc\ntddkbd.h* in the Windows 2000 DDK
- Sample Kbfiltr source code in the *%user's install directory%\src\input\kbfiltr* directory in the Windows 2000 DDK

Kbfiltr Internal Device Control Requests

This section describes the operation of the following internal device control requests:

IOCTL_INTERNAL_I8042_HOOK_KEYBOARD
IOCTL_INTERNAL_KEYBOARD_CONNECT
IOCTL_INTERNAL_KEYBOARD_DISCONNECT

For all other device control requests, Kbfiltr skips the current IRP stack and sends the request down the device stack without further processing.

IOCTL_INTERNAL_I8042_HOOK_KEYBOARD

Operation

The IOCTL_INTERNAL_I8042_HOOK_KEYBOARD request does the following:

- Adds an initialization callback to I8042prt's keyboard initialization routine
- Adds an ISR callback to I8042prt's interrupt service routine

The initialization and ISR callbacks are optional and are provided by an upper-level filter driver for a PS/2-style keyboard device.

After I8042prt receives an IOCTL_INTERNAL_KEYBOARD_CONNECT request, it sends a synchronous IOCTL_INTERNAL_I8042_HOOK_KEYBOARD request to the top of the keyboard device stack.

After Kbfiltr receives the hook keyboard request, Kbfiltr filters the request in the following manner:

- Saves the upper-level information passed to Kbfiltr, which includes the context of an upper-level device object, a pointer to an initialization callback, and a pointer to an ISR callback
- Replaces the upper-level information with its own
- Saves the context of I8042prt and pointers to callbacks that the Kbfiltr ISR callback can use

For more information on this request and the callback routines, see the following topics:

- *Operation of Non-HID Keyboard and Mouse Drivers* in the online DDK
- *I8042prt Callback Routines*
- *Kbfiltr Callback Routines*

Input

The **Parameters.DeviceIoControl.InputBufferLength** member is set to a value greater than or equal to the size in bytes of an INTERNAL_I8042_HOOK_KEYBOARD structure.

The **Parameters.DeviceIoControl.Type3InputBuffer** points to an INTERNAL_I8042_HOOK_KEYBOARD structure. This structure includes the following members:

InitializationRoutine

Pointer to an optional callback routine that is called by I8042prt when it initializes a keyboard device.

IsrRoutine

Pointer to an optional callback routine that is called by the interrupt service routine of I8042prt.

I/O Status Block

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of an INTERNAL_I8042_HOOK_KEYBOARD structure.

IOCTL_INTERNAL_KEYBOARD_CONNECT

Operation

The IOCTL_INTERNAL_KEYBOARD_CONNECT request connects the Kbdclass service to the keyboard device. Kbdclass sends this request down the keyboard device stack before it opens the keyboard device.

After Kbfiltr received the keyboard connect request, Kbfiltr filters the connect request in the following way:

- Saves a copy of Kbdclass's CONNECT_DATA structure that is passed to the filter driver by Kbdclass
- Substitutes its own connect information for the class driver connect information
- Sends the IOCTL_INTERNAL_KEYBOARD_CONNECT request down the device stack

If the request is not successful, Kbfiltr completes the request with an appropriate error status.

Kbfiltr provides a template for a filter service callback that can supplement the operation of **KeyboardClassServiceCallback**, the Kbdclass class service callback. The filter service callback can filter the input data that is transferred from the device input buffer to the class data queue.

For more information on the connection of the Kbdclass service, see the following topics:

- *Connect a Class Service Callback and a Filter Service Callback to a Device* in the online DDK
- *Kbdclass Callback Routines*
- *Kbfiltr Callback Routines*

Input

The **Parameters.DeviceIoControl.InputBufferLength** member is set to a value greater than or equal to the size in bytes of a CONNECT_DATA structure.

The **Parameters.DeviceIoControl.Type3InputBuffer** member points to a CONNECT_DATA structure that is allocated and set by Kbdclass.

Output

The **Parameters.DeviceIoControl.Type3InputBuffer** member points to a CONNECT_DATA structure that is set by Kbfiltr.

I/O Status Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength is less that the size in bytes of a CONNECT_DATA structure.

STATUS_SHARING_VIOLATION

Kbfiltr is already connected (the filter driver supports only one connect request).

IOCTL_INTERNAL_KEYBOARD_DISCONNECT

Operation

The IOCTL_INTERNAL_KEYBOARD_DISCONNECT request is completed with a status of STATUS_NOT_IMPLEMENTED. Note that a Plug and Play keyboard can be added or removed by the Plug and Play Manager.

I/O Status Block

The **Status** member is set to STATUS_NOT_IMPLEMENTED.

Kbfiltr Callback Routines

This section describes the following Kbfiltr callback routines:

- **KbFilter_InitializationRoutine**
- **KbFilter_IsrHook**
- **KbFilter_ServiceCallback**

KbFilter_InitializationRoutine

```
NTSTATUS
  KbFilter_InitializationRoutine(
    IN PDEVICE_OBJECT DeviceObject,
    IN PVOID SynchFuncContext,
    IN PI8042_SYNCH_READ_PORT ReadPort,
    IN PI8042_SYNCH_WRITE_PORT WritePort,
    OUT PBOOLEAN TurnTranslationOn
    );
```

The **KbFilter_InitializationRoutine** routine is a tempate for a keyboard **Initialization-Routine** callback routine that supplements the default initialization of a keyboard device by I8042prt.

Parameters

DeviceObject

Pointer to the device object that is the context for this routine.

SynchFuncContext

Pointer to the context for the routines pointed to by *ReadPort* and *Writeport.*

ReadPort

Pointer to a routine that reads from the port.

WritePort

Pointer to a routine that writes to the port.

TurnTranslationOn

Specifies whether to turn translation on or off. If *TranslationOn* is TRUE, translation is turned on; otherwise, translation is turned off.

Include

kbfiltr.h

Return Value

KbFilter_InitializationRoutine returns an appropriate NTSTATUS code.

Comments

KbFilter_InitializationRoutine is not needed if I8042prt's default initialization of a keyboard is sufficient.

I8042prt calls **KbFilter_InitializationRoutine** when it initializes the keyboard. Default keyboard initialization includes the following operations: reset the keyboard, set the typematic rate and delay, and set the light-emitting diodes (LED).

KbFilter_InitializationRoutine runs in kernel mode at IRQL DISPATCH_LEVEL.

See Also

Keyboard InitializationRoutine

KbFilter_IsrHook

```
KbFilter_IsrHook(
    IN PDEVICE_OBJECT DeviceObject,
    IN PKEYBOARD_INPUT_DATA CurrentInput,
    IN POUTPUT_PACKET CurrentOutput,
    IN OUT UCHAR StatusByte,
    IN PUCHAR DataByte,
    OUT PBOOLEAN ContinueProcessing,
    IN PKEYBOARD_SCAN_STATE ScanState
    );
```

The **KbFilter_IsrHook** routine is a template for a keyboard **IsrRoutine** callback routine that customizes the operation of I8042prt's keyboard ISR.

Parameters

DeviceObject

Pointer to the filter device object of the driver that supplies this callback routine.

CurrentInput

Pointer to the input KEYBOARD_DATA_INPUT structure that is being constructed by the ISR.

CurrentOutput

Pointer to the array of bytes that is being written to the hardware device.

StatusByte

Specifies the status byte that is read from I/O port 60 when an interrupt occurs.

DataByte

Specifies the data byte that is read from I/O port 64 when an interrupt occurs.

ContinueProcessing

Specifies whether processing in the ISR will continue after this routine completes.

ScanState

Specifies the keyboard scan state.

Include

kbfiltr.h

Return Value

KbFilter_IsrHook returns TRUE if the interrupt service routine should continue; otherwise it returns FALSE.

Comments

This callback is not needed if the default operation of I8042prt is sufficient.

I8042prt's keyboard ISR calls **KbFilter_IsrHook** after it validates the interrupt and reads the scan code.

KbFilter_IsrHook runs in kernel mode at the IRQL of I8042prt's keyboard ISR.

KbFilter_ServiceCallback

```
VOID
  KbFilter_ServiceCallback(
    IN PDEVICE_OBJECT DeviceObject,
    IN PKEYBOARD_INPUT_DATA InputDataStart,
    IN PKEYBOARD_INPUT_DATA InputDataEnd,
    IN OUT PULONG InputDataConsumed
    );
```

The **KbFilter_ServiceCallback** routine is a template for a filter service callback that supplements the operation of **KeyboardClassServiceCallback**.

Parameters

DeviceObject

Pointer to the class device object.

InputDataStart

Pointer to the first keyboard data packet (a KEYBOARD_INPUT_DATA structure) in the port device's input buffer.

InputDataEnd

Pointer to the keyboard data packet that immediately follows the last data packet in the port device's input data buffer.

InputDataConsumed

Pointer to the number of keyboard data packets that are transferred by the routine.

Include

kbfiltr.h

Comments

The ISR dispatch completion routine of the function driver calls **KbFilter_Service-Callback**, which then calls **KeyboardClassServiceCallback**. The filter service callback can be configured to modify the input data that is transferred from the device's input buffer to the class data queue. For example, the callback can delete, transform, or insert data.

KbFilter_ServiceCallback runs in kernel mode at IRQL DISPATCH_LEVEL.

CHAPTER 10

Moufiltr Driver Reference

This chapter includes the following topics about *Moufiltr*, the sample upper-level mouse filter driver in the Microsoft® Windows® 2000 DDK:

- *Moufiltr Internal Device Control Requests*
- *Moufiltr Callback Routines*

Moufiltr is designed to be used with *Mouclass*, the Windows 2000 system class driver for mouse devices, and *I8042prt*, the Windows 2000 function driver for a PS/2-style mouse. Moufiltr demonstrates how to filter I/O requests and add callback routines that modify the operation of Mouclass and I8042prt.

See the following topics for more information about Moufiltr operation:

- *Keyboard and Mouse Drivers for Non-HID Devices* in the online DDK
- Include file *%user's install path%\inc\ntddmou.h* in the Windows 2000 DDK
- Sample Moufiltr source code in the *%user's install directory%\src\input\Moufiltr* directory in the Windows 2000 DDK

Moufiltr Internal Device Control Requests

This section describes the operation of the following internal device control requests that Moufiltr supports:

IOCTL_INTERNAL_I8042_HOOK_MOUSE
IOCTL_INTERNAL_MOUSE_CONNECT
IOCTL_INTERNAL_MOUSE_DISCONNECT

For all other requests, Moufiltr skips the current IRP stack and sends the request down the device stack without further processing.

IOCTL_INTERNAL_I8042_HOOK_MOUSE

Operation

The IOCTL_INTERNAL_I8042_HOOK_MOUSE request adds an ISR callback to I8042prt's interrupt service routine for a PS/2-style mouse. The ISR callback is optional and is provided by an upper-level mouse filter driver.

I8042prt sends this request after it receives an *IOCTL_INTERNAL_MOUSE_CONNECT request*. I8042prt sends a synchronous IOCTL_INTERNAL_I8042_HOOK_MOUSE request to the top of the mouse device stack.

After Moufiltr receives the hook mouse request, it filters the request in the following way:

- Saves the upper-level information passed to Moufiltr, which includes the context of an upper-level device object and a pointer to an ISR callback
- Replaces the upper-level information with its own
- Saves the context of I8042prt and pointers to callbacks that the Moufiltr ISR callback routine can use

For more information on this request and the callback routines, see the following topics:

- *Operation of Non-HID Keyboard and Mouse Drivers* in the online DDK
- *I8042prt Mouse Callback Routines*
- *Moufiltr Callback Routines*

Input

The **Parameters.DeviceIoControl.InputBufferLength** member is set to a value greater than or equal to the size in bytes of an INTERNAL_I8042_HOOK_MOUSE structure.

The **Parameters.DeviceIoControl.Type3InputBuffer** points to an INTERNAL_I8042_HOOK_MOUSE structure that is allocated and set initially by I8042prt.

I/O Status Block

The **Status** member is set to one of the following values:

STATUS_SUCCESS

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of an INTERNAL_I8042_HOOK_MOUSE structure.

IOCTL_INTERNAL_MOUSE_CONNECT

Operation

The IOCTL_INTERNAL_MOUSE_CONNECT request connects Mouclass service to a mouse device. Mouclass sends this request down the device stack before it opens a mouse device.

After Moufiltr receives the mouse connect request, it filters the request in the following manner:

- Saves a copy of CONNECT_DATA structure that was passed to Moufiltr
- Substitutes its own connect information for the class driver connect information
- Sends the IOCTL_INTERNAL_MOUSE_CONNECT request down the device stack

If the request is not successful, Moufiltr completes the request with an appropriate error status.

Moufiltr provides a template for a filter service callback that can supplement the operation of **MouseClassServiceCallback**, the Mouclass service callback. The filter service callback can filter the input data that is transferred from the device input buffer to the class driver data queue.

For more information on the connection of the Kbdclass service, see the following topics:

- *Connect a Class Service Callback and a Filter Service Callback to a Device* in the online DDK
- *I8042prt Mouse Callback Routines*
- *Moufiltr Callback Routines*

Input

The **Parameters.DeviceIoControl.InputBufferLength** member is set to a value greater than or equal to the size in bytes of a CONNECT_DATA structure.

The **Parameters.DeviceIoControl.Type3InputBuffer** member points to a CONNECT_DATA structure that is allocated and set by Mouclass.

Output

The **Parameters.DeviceIoControl.Type3InputBuffer** member points to a CONNECT_DATA structure that is set by Moufiltr.

I/O Status Block

The **Information** member is set to zero.

The **Status** member is set to one of the following values:

STATUS_INVALID_PARAMETER

Parameters.DeviceIoControl.InputBufferLength is less than the size in bytes of a CONNECT_DATA structure.

STATUS_SHARING_VIOLATION

Moufiltr is already connected (a filter driver supports only one connect request).

IOCTL_INTERNAL_MOUSE_DISCONNECT

Operation

The IOCTL_INTERNAL_MOUSE_DISCONNECT request is completed by Moufiltr with an error status of STATUS_NOT_IMPLEMENTED. (Note that a Plug and Play mouse device can be added or removed by the Plug and Play Manager.)

I/O Status Block

The **Status** member is set to STATUS_NOT_IMPLEMENTED.

Moufiltr Callback Routines

This section describes the following Moufiltr callback routines:

- **MouFilter_IsrHook**
- **MouFilter_ServiceCallback**

MouFilter_IsrHook

```
BOOLEAN
  MouFilter_IsrHook(
    IN PDEVICE_OBJECT DeviceObject,
    IN PMOUSE_INPUT_DATA CurrentInput,
    IN POUTPUT_PACKET CurrentOutput,
    IN UCHAR StatusByte,
    IN OUT PUCHAR DataByte,
    OUT PBOOLEAN ContinueProcessing,
    IN PMOUSE_STATE MouseState,
    IN PMOUSE_RESET_SUBSTATE ResetSubState
    );
```

MouFilter_IsrHook is a template for a mouse **IsrRoutine** callback routine that customizes the operation of I8042prt's mouse ISR.

Parameters

DeviceObject

Pointer to the filter device object of the driver that supplies this callback routine.

CurrentInput

Pointer to the input MOUSE_DATA_INPUT structure being constructed by the ISR.

CurrentOutput

Pointer to the array of bytes being written to the hardware device.

StatusByte

Specifies a status byte that is read from I/O port 60 when the interrupt occurs.

DataByte

Specifies a data byte that is read from I/O port 64 when the interrupt occurs.

ContinueProcessing

Specifies a flag indicating whether processing in the ISR will continue after this routine completes.

MouseState

Specifies the mouse state.

ResetSubState

Specifies a mouse substate.

Include

moufiltr.h

Return Value

MouFilter_IsrHook returns TRUE if the interrupt service routine should continue; otherwise it returns FALSE.

Comments

A **MouFilter_IsrHook** callback routine is not needed if the default operation of I8042prt is sufficient.

I8042prt's ISR calls **MouFilter_IsrHook** after it validates the interrupt.

MouFilter_IsrHook runs in kernel mode at the IRQL of I8042prt's ISR for mouse device.

MouFilter_ServiceCallback

```
VOID
  MouFilter_ServiceCallback(
    IN PDEVICE_OBJECT DeviceObject,
    IN PMOUSE_INPUT_DATA InputDataStart,
    IN PMOUSE_INPUT_DATA InputDataEnd,
    IN OUT PULONG InputDataConsumed
    );
```

MouFilter_ServiceCallback is a template for a filter service callback that supplements **MouseClassServiceCallback**.

Parameters

DeviceObject

Pointer to the class device object.

InputDataStart

Pointer to the first *mouse data packet* (a MOUSE_INPUT_DATA structure) in the input data buffer of the port device.

InputDataEnd

Pointer to the mouse data packet immediately following the last mouse data packet in the port device's input data buffer.

InputDataConsumed

Pointer to the number of mouse data packets that are transferred by the routine.

Include

moufiltr.h

Comments

The ISR DPC of I8042prt calls **MouFilter_ServiceCallback**, which then calls **MouseClassServiceCallback**. A filter service callback can be configured to modify the input data that is transferred from the device's input buffer to the class data queue. For example, the callback can delete, transform, or insert data.

MouFilter_ServiceCallback runs in kernel mode at IRQL DISPATCH_LEVEL.

PART 4

USB Drivers

CHAPTER 1

I/O Requests for USB Client Drivers

Windows Driver Model (WDM) clients of the Universal Serial Bus (USB) driver stack communicate to the USB driver stack, by submitting an IRP with major code IRP_MJ_INTERNAL_DEVICE_CONTROL, and one of the following minor codes:

IOCTL_INTERNAL_USB_SUBMIT_URB
IOCTL_INTERNAL_USB_GET_PORT_STATUS
IOCTL_INTERNAL_USB_RESET_PORT
IOCTL_INTERNAL_USB_GET_ROOTHUB_PDO
IOCTL_INTERNAL_USB_ENABLE_PORT
IOCTL_INTERNAL_USB_GET_HUB_COUNT
IOCTL_INTERNAL_USB_CYCLE_PORT
IOCTL_INTERNAL_USB_GET_HUB_NAME
IOCTL_INTERNAL_USB_GET_BUS_INFO
IOCTL_INTERNAL_USB_GET_CONTROLLER_NAME

IOCTL_INTERNAL_USB_SUBMIT_URB

Drivers can use this request to submit an URB to the bus driver.

Input

Parameters.Others.Argument1 points to the URB, a variable-length structure. The **UrbHeader.Function** member of the URB specifies the URB type, and the **UrbHeader.Length** member specifies the size in bytes of the URB. The length of URB, as well as the meaning of any additional members depends on the value of **UrbHeader.Function**. See *URB* in Chapter 3 for details.

Output

Parameters.Others.Argument1 points to the URB structure. The **UrbHeader.Status** contains a USB status code for the requested operation. Any additional output depends on the **UrbHeader.Function** member of the URB submitted. See *URB* for details.

I/O Status Block

The lower-level drivers will set **Irp->IoStatus.Status** to STATUS_SUCCESS if the URB can be successfully processed. Otherwise, the bus driver will set it to the appropriate error condition, such as STATUS_INVALID_PARAMETER, or STATUS_INSUFFICIENT_RESOURCES.

See Also

URB

IOCTL_INTERNAL_USB_RESET_PORT

Drivers can use this request to reset the port associated to a PDO. This IOCTL must be sent at an IRQL of PASSIVE_LEVEL.

Input

None.

Output

None.

I/O Status Block

The bus or port driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or the appropriate error status.

IOCTL_INTERNAL_USB_GET_PORT_STATUS

This I/O request queries the status of the PDO. This IOCTL must be sent at an IRQL of PASSIVE_LEVEL.

Input

Parameters.Others.Argument1 should be a pointer to a ULONG to be filled in with the port status flags.

Output

Parameters.Others.Argument1 points to a ULONG that has the port status flags filled in. The flags can be one or both of USBP_PORT_ENABLED, USBP_PORT_CONNECTED.

I/O Status Block

The bus or port driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or the appropriate error status.

IOCTL_INTERNAL_USB_ENABLE_PORT

This I/O request re-enables the port associated with a PDO. This IOCTL must be sent at an IRQL of PASSIVE_LEVEL.

Input

None.

Output

None.

I/O Status Block

The bus or port driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or the appropriate error status.

Operation

This request may fail if the device is attached to a hub other than the root hub. In that case, a driver may use IOCTL_INTERNAL_USB_RESET_PORT to re-enable the port.

IOCTL_INTERNAL_USB_GET_HUB_COUNT

This I/O request is used internally by the hub driver. Do not use this request.

IOCTL_INTERNAL_USB_CYCLE_PORT

This I/O request simulates a device unplug and re-plug on the port associated with the PDO. This IOCTL must be sent at an IRQL of PASSIVE_LEVEL.

Input

None.

Output

None.

I/O Status Block

The bus or port driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or the appropriate error status.

IOCTL_INTERNAL_USB_GET_ROOTHUB_PDO

This I/O request is used internally by the hub driver. Do not use this request.

IOCTL_INTERNAL_USB_GET_HUB_NAME

Drivers can use this request to get the device name of the USB hub. This IOCTL must be sent at an IRQL of PASSIVE_LEVEL.

Input

Parameters.DeviceIoControl.OutputBufferLength is the length of the buffer (in bytes) passed in the **Irp->AssociatedIrp.SystemBuffer** field.

Irp->AssociatedIrp.SystemBuffer points to a USB_ROOT_HUB_NAME structure.

Output

Irp->AssociatedIrp.SystemBuffer is filled with the root hub's symbolic name.

I/O Status Block

A lower-level driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or the appropriate error status. It will set **Irp->IoStatus.Information** to number of bytes required to hold the USB_ROOT_HUB_NAME structure. If the request fails, the driver can use this information to resubmit the request with a big enough buffer.

See Also

USB_ROOT_HUB_NAME

IOCTL_INTERNAL_USB_GET_BUS_INFO

This I/O request queries the bus driver for certain bus information. This IOCTL must be sent at an IRQL of PASSIVE_LEVEL.

Input

Parameters.Others.Argument1 should be a pointer to a PUSB_BUS_NOTIFICATION structure.

Output

Parameters.Others.Argument1 points to a PUSB_BUS_NOTIFICATION structure that has the **TotalBandwidth**, **ConsumedBandwidth**, and **ControllerNameLength** fields filled in.

I/O Status Block

The bus or port driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or the appropriate error status.

IOCTL_INTERNAL_USB_GET_CONTROLLER_NAME

This I/O request queries the bus driver for the device name of the USB host controller. This IOCTL must be sent at an IRQL of PASSIVE_LEVEL.

Input

Parameters.Others.Argument1 should be a pointer to a USB_HUB_NAME structure that will be filled in with the name of the host controller.

Parameters.Others.Argument2 should be a ULONG specifying the length of the buffer (in bytes) in **Parameters.Others.Argument1**.

Output

The bus driver will fill the buffer pointed to by **Parameters.Others.Argument2** with the host controller device name. It will be filled only up to the length specified in **Parameters. Others.Argument1**.

I/O Status Block

The bus or port driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or the appropriate error status.

See Also

USB_HUB_NAME

CHAPTER 2

USB Client Support Routines

Universal Serial Bus (USB) client drivers can call the following routines on Windows Driver Model (WDM) platforms. These routines assist a client driver in communicating with the USB driver stack.

Routines are described in alphabetical order.

GET_ISO_URB_SIZE

```
ULONG
  GET_ISO_URB_SIZE(
    IN ULONG NumberOfPackets
    );
```

GET_ISO_URB_SIZE returns the number of bytes required to hold an isochronous transfer request.

Parameters

NumberOfPackets

Specifies the number of isochronous transfer packets that will be part of the transfer request.

Return Value

GET_ISO_URB_SIZE returns the number of bytes required to hold an isochronous request with the given *NumberOfPackets*.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

URB, _URB_ISOCH_TRANSFER, USBD_ISO_PACKET_DESCRIPTOR

GET_SELECT_CONFIGURATION_REQUEST_SIZE

```
ULONG
  GET_SELECT_CONFIGURATION_REQUEST_SIZE(
    IN ULONG TotalInterfaces,
    IN ULONG TotalPipes
    );
```

GET_SELECT_CONFIGURATION_REQUEST_SIZE returns the number of bytes required to create a select configuration URB.

Parameters

TotalInterfaces

Specifies how many interfaces the configuration has.

TotalPipes

Specifies how many endpoints (pipes) the configuration has.

Return Value

GET_SELECT_CONFIGURATION_REQUEST_SIZE returns the number of bytes required to hold a select configuration request with the given number of pipes and interfaces.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

URB, **UsbBuildSelectConfigurationRequest**, **USBD_CreateConfigurationRequest**

GET_SELECT_INTERFACE_REQUEST_SIZE

```
ULONG
  GET_SELECT_INTERFACE_REQUEST_SIZE(
    IN ULONG TotalPipes
    );
```

GET_SELECT_INTERFACE_REQUEST_SIZE returns the number of bytes required to create a select interface URB.

Parameters

TotalPipes

Specifies the total number of endpoints (pipes) the interface has.

Return Value

GET_SELECT_INTERFACE_REQUEST_SIZE returns the number of bytes required to hold an URB to select a new setting for an interface with the given number of pipes.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

URB, **UsbBuildSelectInterfaceRequest**

GET_USBD_INTERFACE_SIZE

```
ULONG
  GET_USBD_INTERFACE_SIZE(
    IN ULONG TotalEndpoints
    );
```

GET_USBD_INTERFACE_SIZE returns the number of bytes required to hold a USBD_INTERFACE_INFORMATION interface descriptor with its associated endpoint descriptors.

Parameters

TotalEndpoints

Specifies the total number of endpoints (pipes) the interface has.

Return Value

GET_USBD_INTERFACE_SIZE returns the number of bytes required to hold a USBD_INTERFACE_INFORMATION structure describing the interface and a USBD_PIPE_INFORMATION structure for each endpoint in the interface.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

USBD_INTERFACE_INFORMATION, USBD_PIPE_INFORMATION

UsbBuildFeatureRequest

```
VOID
  UsbBuildFeatureRequest(
    IN OUT PURB Urb,
    IN USHORT Op,
    IN USHORT FeatureSelector,
    IN USHORT Index,
    IN PURB Link    OPTIONAL
    );
```

UsbBuildFeatureRequest formats an URB with the parameters necessary to request that a feature be turned on or off on a USB device.

Parameters

Urb

Points to an URB to be formatted as a feature request to a device.

Op

Specifies one of the following operation codes:

URB_FUNCTION_SET_FEATURE_TO_DEVICE

Sets a USB-defined feature, specified by *FeatureSelector*, on a device.

URB_FUNCTION_SET_FEATURE_TO_INTERFACE

Sets a USB-defined feature, specified by *FeatureSelector*, on an interface for a device.

URB_FUNCTION_SET_FEATURE_TO_ENDPOINT

Sets a USB-defined feature, specified by *FeatureSelector*, on an endpoint for an interface on a USB device.

URB_FUNCTION_SET_FEATURE_TO_OTHER

Sets a USB-defined feature, specified by *FeatureSelector*, on a device-defined target on a USB device.

URB_FUNCTION_CLEAR_FEATURE_TO_DEVICE

Clears a USB-defined feature, specified by *FeatureSelector*, on a device.

URB_FUNCTION_CLEAR_FEATURE_TO_INTERFACE

Clears a USB-defined feature, specified by *FeatureSelector*, on an interface for a device.

URB_FUNCTION_CLEAR_FEATURE_TO_ENDPOINT

Clears a USB-defined feature, specified by *FeatureSelector*, on an endpoint, for an interface, on a USB device.

URB_FUNCTION_CLEAR_FEATURE_TO_OTHER

Clears a USB-defined feature, specified by *FeatureSelector*, on a device defined target on a USB device.

FeatureSelector

Specifies the USB-defined feature code that should be set or cleared on the target as specified by *Op*.

Index

For a feature request for an endpoint or interface, specifies the index of the endpoint or interface within the configuration descriptor. For the device, this must be zero.

Link

Points to an caller-initialized URB. *Link* becomes the subsequent URB in a chain of requests with *Urb* being its predecessor.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

URB, _URB_CONTROL_FEATURE_REQUEST

UsbBuildGetDescriptorRequest

```
VOID
  UsbBuildGetDescriptorRequest(
    IN OUT PURB Urb,
    IN USHORT Length,
    IN UCHAR DescriptorType,
    IN UCHAR Index,
    IN USHORT LanguageId,
    IN PVOID TransferBuffer    OPTIONAL,
    IN PMDL TransferBufferMDL    OPTIONAL,
    IN ULONG TransferBufferLength,
    IN PURB Link    OPTIONAL
    );
```

UsbBuildGetDescriptorRequest formats an URB with the parameters necessary to obtain descriptor information from the host controller driver (HCD).

Parameters

Urb

Points to an URB to be formatted for a get descriptor request to the HCD. The caller must allocate nonpaged pool for this URB.

Length

Specifies the size, in bytes, of the URB.

DescriptorType

Specifies one of the following values:

USB_DEVICE_DESCRIPTOR_TYPE
USB_CONFIGURATION_DESCRIPTOR_TYPE
USB_STRING_DESCRIPTOR_TYPE

Index

Specifies the device-defined index of the descriptor that is to be retrieved.

LanguageId

Specifies the language ID of the descriptor to be retrieved when USB_STRING_DESCRIPTOR_TYPE is set in *DescriptorType*. This parameter must be 0 for any other value in *DescriptorType*.

TransferBuffer

Points to a resident buffer to receive the descriptor data or is NULL if an MDL is supplied in *TransferBufferMDL*.

TransferBufferMdl

Points to an MDL that describes a resident buffer to receive the descriptor data or is NULL if a buffer is supplied in *TransferBuffer*.

TransferBufferLength

Specifies the length of the buffer specified in *TransferBuffer* or described in *TransferBufferMDL*.

Link

Points to an caller-initialized URB. *Link* becomes the subsequent URB in a chain of requests with *Urb* being its predecessor.

Comments

When USB_CONFIGURATION_DESCRIPTOR_TYPE is specified for *DescriptorType*, all interface, endpoint, class-specific, and vendor-specific descriptors for the configuration also are retrieved. The caller must allocate a buffer large enough to hold all of this information or the data is truncated without error.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

URB, USB_DEVICE_DESCRIPTOR

UsbBuildGetStatusRequest

```
VOID
  UsbBuildGetStatusRequest(
    IN OUT Urb,
    IN USHORT Op,
    IN USHORT Index,
    IN PVOID TransferBuffer    OPTIONAL,
    IN PMDL TransferBufferMDL,
    IN PURB Link
    );
```

UsbBuildGetStatusRequest formats an URB to obtain status from a device, interface, endpoint, or other device-defined target on a USB device.

Parameters

Urb

Points to an URB to be formatted as an status request.

Op

Specifies one of the following values:

URB_FUNCTION_GET_STATUS_FROM_DEVICE

Retrieves status from a USB device.

URB_FUNCTION_GET_STATUS_FROM_INTERFACE

Retrieves status from an interface on a USB device.

URB_FUNCTION_GET_STATUS_FROM_ENDPOINT

Retrieves status from an endpoint for an interface on a USB device.

URB_FUNCTION_GET_STATUS_FROM_OTHER

Retrieves status from a device-defined target on a USB device.

Index

Specifies the device-defined index, returned by a successful configuration request, if the request is for an endpoint or interface. Otherwise, *Index* must be zero.

TransferBuffer

Points to a resident buffer to receive the status data or is NULL if an MDL is supplied in *TransferBufferMDL*.

TransferBufferMDL

Points to an MDL that describes a resident buffer to receive the status data or is NULL if a buffer is supplied in *TransferBuffer*.

Link

Points to an caller-initialized URB. *Link* becomes the subsequent URB in a chain of requests with *Urb* being its predecessor.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

URB, _URB_CONTROL_GET_STATUS_REQUEST

UsbBuildInterruptOrBulkTransferRequest

```
VOID
  UsbBuildGetInterruptOrBulkTransferRequest(
    IN OUT PURB Urb,
    IN USHORT Length,
    IN USBD_PIPE_HANDLE PipeHandle,
    IN PVOID TransferBuffer    OPTIONAL,
    IN PMDL TransferBufferMDL    OPTIONAL,
    IN ULONG TransferBufferLength,
    IN ULONG TransferFlags,
    IN PURB Link
    );
```

UsbBuildInterruptOrBulkTransferRequest formats an URB to send or receive data on a bulk pipe, or to receive data from an interrupt pipe.

Parameters

Urb

Points to an URB to be formatted as an interrupt or bulk transfer request.

Length

Specifies the size, in bytes, of the URB.

PipeHandle

Specifies the handle for this pipe returned by the HCD when a configuration was selected.

TransferBuffer

Points to a resident buffer for the transfer or is NULL if an MDL is supplied in *TransferBufferMDL*. The contents of this buffer depend on the value of *TransferFlags*. If USBD_TRANSFER_DIRECTION_IN is specified, this buffer will contain data read from the device on return from the HCD. Otherwise, this buffer contains driver-supplied data to be transferred to the device.

TransferBufferMdl

Points to an MDL that describes a resident buffer or is NULL if a buffer is supplied in *TransferBuffer*. The contents of the buffer depend on the value of *TransferFlags*. If USBD_TRANSFER_DIRECTION_IN is specified, the described buffer will contain data read from the device on return from the HCD. Otherwise, the buffer contains driver-supplied data to be transferred to the device. The MDL must be allocated from nonpaged pool.

TransferBufferLength

Specifies the length, in bytes, of the buffer specified in *TransferBuffer* or described in *TransferBufferMDL*.

TransferFlags

Specifies zero, one, or a combination of the following flags:

USBD_TRANSFER_DIRECTION_IN

Is set to request data from a device. To transfer data to a device, this flag must be clear. The flag must be set if the pipe is an interrupt transfer pipe.

USBD_SHORT_TRANSFER_OK

Can be used if USBD_TRANSFER_DIRECTION_IN is set. If set, directs the HCD not to return an error if a packet is received from the device that is shorter than the maximum packet size for the endpoint. Otherwise, a short request is returns an error condition.

Link

Points to an caller-initialized URB. *Link* becomes the subsequent URB in a chain of requests with *Urb* being its predecessor.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

URB, USB_DEVICE_DESCRIPTOR

UsbBuildSelectConfigurationRequest

```
VOID
  UsbBuildSelectConfigurationRequest(
    IN PURB Urb,
    IN USHORT Length,
    IN PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor
    );
```

UsbBuildSelectConfigurationRequest formats an URB with the parameters necessary to select a configuration on a USB device.

Parameters

Urb

Points to an URB to be formatted as a select configuration request.

Length

Specifies the size, in bytes, of the URB.

ConfigurationDescriptor

Points to an initialized USB configuration descriptor that identifies the configuration to be set on the device. If NULL, the device will be set into its unconfigured state.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

URB, _URB_SELECT_CONFIGURATION, USB_CONFIGURATION_DESCRIPTOR

UsbBuildSelectInterfaceRequest

```
VOID
  UsbBuildSelectInterfaceRequest(
    IN PURB Urb,
    IN USHORT Length,
    IN USBD_CONFIGURATION_HANDLE ConfigurationHandle,
    IN UCHAR InterfaceNumber,
    IN UCHAR AlternateSetting
    );
```

UsbBuildSelectInterfaceRequest formats an URB with the parameters necessary to select an alternate setting for an interface on a USB device.

Parameters

Urb

Points to an URB that is to be formatted as a select interface request.

Length

Specifies the size, in bytes, of the URB. The URB_FUNCTION_SELECT_INTERFACE URB has a variable length. Clients can use the GET_SELECT_INTERFACE_REQUEST_SIZE macro to determine the URB length.

ConfigurationHandle

Specifies the handle for this interface returned by the HCD when a configuration was selected.

InterfaceNumber

Is the device-defined identifier for this interface specified in the descriptor for this interface.

AlternateSetting

Is the device-defined identifier of the alternate setting that this interface should now use.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

URB, _URB_SELECT_INTERFACE

UsbBuildVendorRequest

```
VOID
  UsbBuildVendorRequest(
    IN PURB Urb,
    IN USHORT Function,
    IN USHORT Length,
    IN ULONG TransferFlags,
    IN UCHAR ReservedBits,
    IN UCHAR Request,
    IN USHORT Value,
    IN USHORT Index,
    IN PVOID TransferBuffer    OPTIONAL,
    IN PMDL TransferBufferMDL    OPTIONAL,
    IN ULONG TransferBufferLength,
    IN PURB Link    OPTIONAL,
    );
```

UsbBuildVendorRequest formats an URB to send a vendor or class-specific command to a USB device, interface, endpoint, or other device-defined target.

Parameters

Urb

Points to an URB that is to be formatted as a vendor or class request.

Function

Must be set to one of the following values:

URB_FUNCTION_VENDOR_DEVICE

Indicates the URB is a vendor-defined request for a USB device.

URB_FUNCTION_VENDOR_INTERFACE

Indicates the URB is a vendor-defined request for an interface on a USB device.

URB_FUNCTION_VENDOR_ENDPOINT

Indicates the URB is a vendor-defined request for an endpoint, in an interface, on a USB device.

URB_FUNCTION_VENDOR_OTHER

Indicates the URB is a vendor-defined request for a device-defined target.

URB_FUNCTION_CLASS_DEVICE

Indicates the URB is a USB-defined class request for a USB device.

URB_FUNCTION_CLASS_INTERFACE

Indicates the URB is a USB-defined class request for an interface on a USB device.

URB_FUNCTION_CLASS_ENDPOINT

Indicates the URB is a USB-defined class request for an endpoint, in an interface, on a USB device.

URB_FUNCTION_CLASS_OTHER

Indicates the URB is a USB-defined class request for a device-defined target.

Length

Specifies the length, in bytes, of the URB.

TransferFlags

Specifies zero, one, or a combination of the following flags:

USBD_TRANSFER_DIRECTION_IN

Is set to request data from a device. To transfer data to a device, this flag must be clear. The flag must be set if the pipe is an interrupt transfer pipe.

USBD_SHORT_TRANSFER_OK

Can be used if USBD_TRANSFER_DIRECTION_IN is set. If set, directs the HCD not to return an error if a packet is received from the device that is shorter than the maximum packet size for the endpoint. Otherwise, a short request is returns an error condition.

ReservedBits

Specifies a value, from 4 to 31 inclusive, that becomes part of the request type code in the USB-defined setup packet. This value is defined by USB for a class request or the vendor for a vendor request.

Request

Specifies the USB or vendor-defined request code for the device, interface, endpoint, or other device-defined target.

Value

Is a value, specific to *Request*, that becomes part of the USB-defined setup packet for the target. This value is defined by the creator of the code used in *Request*.

Index

Specifies the device-defined identifier if the request is for an endpoint, interface, or device-defined target. Otherwise, *Index* must be 0.

TransferBuffer

Points to a resident buffer for the transfer or is NULL if an MDL is supplied in *TransferBufferMDL*. The contents of this buffer depend on the value of *TransferFlags*. If USBD_TRANSFER_DIRECTION_IN is specified, this buffer will contain data read from the device on return from the HCD. Otherwise, this buffer contains driver-supplied data to be transferred to the device.

TransferBufferMdl

Points to an MDL that describes a resident buffer or is NULL if a buffer is supplied in *TransferBuffer*. The contents of the buffer depend on the value of *TransferFlags*. If USBD_TRANSFER_DIRECTION_IN is specified, the described buffer will contain data read from the device on return from the HCD. Otherwise, the buffer contains driver-supplied data to be transferred to the device. The MDL must be allocated from nonpaged pool.

TransferBufferLength

Specifies the length, in bytes, of the buffer specified in *TransferBuffer* or described in *TransferBufferMDL*.

Link

Points to an caller-initialized URB. *Link* becomes the subsequent URB in a chain of requests with *Urb* being its predecessor.

Comments

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

URB, _URB_CONTROL_VENDOR_OR_CLASS_REQUEST

USBD_CreateConfigurationRequest

```
PURB
  USBD_CreateConfigurationRequest(
    IN PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,
    IN OUT PUSHORT Siz
    );
```

This routine is exported to support existing driver binaries and is obsolete. Use **USBD_CreateConfigurationRequestEx** instead.

USBD_CreateConfigurationRequestEx

```
PURB
  USBD_CreateConfigurationRequestEx(
    IN PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,
    IN PUSBD_INTERFACE_LIST_ENTRY InterfaceList
    );
```

USBD_CreateConfigurationRequestEx allocates and formats an URB to select a configuration for a USB device.

Parameters

ConfigurationDescriptor

Points to a configuration descriptor, that includes all interface, endpoint, vendor, and class-specific descriptors, retrieved from a USB device.

InterfaceList

Points to the first element, in a variable-length array of the following structures, that describes interfaces to be made part of the configuration request:

```
typedef struct _USBD_INTERFACE_LIST_ENTRY {
    PUSB_INTERFACE_DESCRIPTOR InterfaceDescriptor;
    PUSBD_INTERFACE_INFORMATION Interface;
} USBD_INTERFACE_LIST_ENTRY, *PUSBD_INTERFACE_LIST_ENTRY;
```

Members

InterfaceDescriptor

Points to a interface descriptor returned from the device as part of a configuration descriptor.

Interface

Points to memory containing information about the interface and all of the endpoints associated with that interface.

Return Value

USBD_CreateConfigurationRequestEx returns an URB that can be sent to the host controller driver (HCD) to set the device in a configured state. The routine allocates memory for this URB that the caller must free when finished with the URB.

Comments

If an interface descriptor is returned in a configuration descriptor, but the caller does not include an entry in the array pointed to by *InterfaceList*, the HCD will ignore that interface and will not initialize the interface.

Before the caller submits the URB returned by this routine, it can override the default settings for the interface(s) or endpoint(s) contained in the interface information structure(s). *InterfaceList[x]->***Interface**, filled in on return from **USBD_CreateConfigurationRequestEx**, points to a USBD_INTERFACE_INFORMATION structure. This structure contains members that can select alternate interface and endpoint settings at device-configuration time. See USBD_INTERFACE_INFORMATION for details on these members.

After all operations with this URB have been completed, the caller must free the memory allocated by this routine for the URB. Failure to do so will result in a memory leak condition.

Callers of this routine can be running at IRQL <= DISPATCH_LEVEL if the memory pointed to by *ConfigurationDescriptor* and *InterfaceList* are allocated from nonpaged pool. Otherwise, callers must be running at IRQL < DISPATCH_LEVEL.

See Also

USBD_INTERFACE_INFORMATION, _URB_SELECT_CONFIGURATION

USBD_GetInterfaceLength

```
ULONG
  USBD_GetInterfaceLength(
    IN PUSB_INTERFACE_DESCRIPTOR InterfaceDescriptor,
    IN PUCHAR BufferEnd
    );
```

USBD_GetInterfaceLength obtains the length of a given interface descriptor, including the length of all endpoint descriptors contained within the interface.

Parameters

InterfaceDescriptor

Points to a interface descriptor for which to obtain the length.

BufferEnd

Points to the position within a buffer at which to stop searching for the length of the interface and associated endpoints.

Return Value

USBD_GetInterfaceLength returns the length, in bytes, of the interface descriptor and all associated endpoint descriptors contained within the interface.

Comments

Callers can use this routine to obtain the length of an interface and associated endpoints that are contained within another buffer. For example, a caller could specify a location inside of a larger buffer for *InterfaceDescriptor* and the beginning of a location of another interface descriptor for *BufferEnd*. This will cause the routine to search only from the beginning of the interface descriptor specified by *InterfaceDescriptor* until either it finds another interface descriptor or it reaches the position specified by *End*.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

USBD_GetUSBDIVersion

```
VOID
  USBD_GetUSBDIVersion(
    OUT PUSBD_VERSION_INFORMATION VersionInformation
    );
```

USBD_GetUSBDIVersion returns version information about the host controller driver (HCD) that controls the clients USB device.

Parameters

VersionInformation

Points to caller-allocated memory for the following structure, that on return from the routine, contains version information about the HCD:

```
typedef struct _USBD_VERSION_INFORMATION {
    ULONG USBDI_Version;
    ULONG Supported_USB_Version;
} USBD_VERSION_INFORMATION, *PUSBD_VERSION_INFORMATION;
```

Members

USBDI_Version

Specifies the version, as a binary-coded decimal (BCD) number, of the HCD.

Supported_USB_Version

Specifies the revision, as a BCD number, of the USB specification that this version of the HCD supports.

Comments

Callers of this routine can be running at IRQL <= DISPATCH_LEVEL if the memory for *VersionInformation* is allocated from nonpaged pool. Otherwise, callers must be running at IRQL < DISPATCH_LEVEL.

USBD_ParseConfigurationDescriptor

```
PUSB_INTERFACE_DESCRIPTOR
  USBD_ParseConfigurationDescriptor(
    IN PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,
    IN UCHAR InterfaceNumber,
    IN UCHAR AlternateSetting
    );
```

This routine is exported to support existing driver binaries and is obsolete. Use **USBD_ ParseConfigurationDescriptorEx** instead.

USBD_ParseConfigurationDescriptorEx

```
PUSB_INTERFACE_DESCRIPTOR
  USBD_ParseConfigurationDescriptorEx(
    IN PUSB_CONFIGURATION_DESCRIPTOR ConfigurationDescriptor,
    IN PVOID StartPosition,
    IN LONG InterfaceNumber,
    IN LONG AlternateSetting,
    IN LONG InterfaceClass,
    IN LONG InterfaceSubClass,
    IN LONG InterfaceProtocol
    );
```

USBD_ParseConfigurationDescriptorEx searches a given configuration descriptor and returns a pointer to an interface that matches the given search criteria.

Parameters

ConfigurationDescriptor

Points to a USB configuration descriptor that contains the interface for which to search.

StartPosition

Points to the address within the configuration descriptor, provided at *Configuration-Descriptor,* to begin searching from. To search from the beginning of the configuration descriptor, the parameters *ConfigurationDescriptor* and *StartPosition* must be the same address.

InterfaceNumber

Specifies the device-defined index of the interface to be retrieved. This should be set to –1 if it should not be a search criteria.

AlternateSetting

Specifies the device-defined alternate-setting index of the interface to be retrieved. If the caller does not wish the alternate setting value to be a search criteria, this parameter should be set to –1.

InterfaceClass

Specifies the device- or USB-defined identifier for the interface class of the interface to be retrieved. If the caller does not wish the interface class value to be a search criteria, this parameter should be set to –1.

InterfaceSubClass

Specifies the device- or USB-defined identifier for the interface subclass of the interface to be retrieved. If the caller does not wish the interface subclass value to be a search criteria, this parameter should be set to –1.

InterfaceProtocol

Specifies the device- or USB-defined identifier for the interface protocol of the interface to be retrieved. If the caller does not wish the interface protocol value to be a search criteria, this parameter should be set to –1.

Return Value

USBD_ParseConfigurationDescriptorEx returns a pointer to the first interface descriptor that matches the given search criteria. If no interface matches the search criteria, it returns NULL.

Comments

Callers can specify more than one of the search criteria (*InterfaceNumber, AlternateSetting, InterfaceClass, InterfaceSubClass,* and *InterfaceProtocol)* when using this routine to find an interface within a configuration descriptor.

When this routine parses the configuration descriptor looking for the interface descriptor that matches the search criteria, it returns the first match, terminating the search. Callers should specify as many search criteria as are necessary to find the desired interface.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL.

See Also

USB_CONFIGURATION_DESCRIPTOR

USBD_ParseDescriptors

```
PUSB_COMMON_DESCRIPTOR
  USBD_ParseDescriptors(
    IN PVOID DescriptorBuffer,
    IN ULONG TotalLength,
    IN PVOID StartPosition,
    IN LONG DescriptorType
    );
```

USBD_ParseDescriptors searches a given configuration descriptor and returns a pointer to the first descriptor that matches the search criteria.

Parameters

DescriptorBuffer

Points to a configuration descriptor that contains the descriptor for which to search.

TotalLength

Specifies the size, in bytes, of the buffer pointed to by *DescriptorBuffer*.

StartPosition

Points to the address within the configuration descriptor, provided at *DescriptorBuffer*, to begin searching from. To search from the beginning of the configuration descriptor, the parameters *DescriptorBuffer* and *StartPosition* must be the same address.

DescriptorType

Specifies the descriptor type code as assigned by USB. The following values are valid for USB-defined descriptor types:

USB_STRING_DESCRIPTOR_TYPE

Specifies that the descriptor being searched for is a string descriptor.

USB_INTERFACE_DESCRIPTOR_TYPE

Specifies that the descriptor being searched for is an interface descriptor.

USB_ENDPOINT_DESCRIPTOR_TYPE

Specifies that the descriptor being searched for is an endpoint descriptor.

Return Value

USBD_ParseDescriptors returns a pointer to the following structure that is the head of the first descriptor that matches the given search criteria, or NULL is returned if no match is found:

```
typedef struct _USB_COMMON_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
} USB_COMMON_DESCRIPTOR, *PUSB_COMMON_DESCRIPTOR;
```

Members

bLength

Specifies the entire length of the descriptor, not this structure.

bDescriptorType

Specifies the descriptor type code, as assigned by USB, for this descriptor.

Comments

This structure is used to hold a portion of a descriptor, so that the caller of **USBD_ParseDescriptors** can determine the correct structure to use to access the remaining data in the descriptor. Every descriptor type has these fields at the beginning of the data and callers can use the **bLength** and **bDescriptorType** members to correctly identify the type of this descriptor.

Comments

When this routine parses the configuration descriptor looking for the descriptor that matches the search criteria, it returns the first match, terminating the search.

Callers of this routine can be running at IRQL <= DISPATCH_LEVEL.

USBD_RegisterHcFilter

```
VOID
  USBD_RegisterHcFilter(
    PDEVICE_OBJECT DeviceObject,
    PDEVICE_OBJECT FilterDeviceObject
    );
```

USBD_RegisterHcFilter is called by USB bus filter drivers to register their device objects with the host controller driver.

Parameters

DeviceObject

Points to the device object that is the current top of the stack as reported by **IoAttach-DeviceToDeviceStack**.

FilterDeviceObject

Points to the filter device object created by the filter driver for its operations.

Comments

USB bus filter drivers must call this routine after attaching their device object to the device object stack for the host controller driver.

Callers of this routine must be running at IRQL PASSIVE_LEVEL.

See Also

IoAttachDeviceToDeviceStack

CHAPTER 3

USB Structures

This chapter describes system-defined structures used by Universal Serial Bus (USB) client drivers on Windows® Driver Model (WDM) platforms. See Part 1 for information about system-defined structures that are not described here.

Drivers can use only those members of structures that are described here. All undocumented members of these structures are reserved for system use.

Transfer buffers that are members of structures defined here must be nonpageable memory.

Structures described in this chapter are in alphabetical order.

URB

```
typedef struct _URB {
  union {
    struct _URB_HEADER UrbHeader;
    struct _URB_SELECT_INTERFACE UrbSelectInterface;
    struct _URB_SELECT_CONFIGURATION UrbSelectConfiguration;
    struct _URB_PIPE_REQUEST UrbPipeRequest;
    struct _URB_FRAME_LENGTH_CONTROL UrbFrameLengthControl;
    struct _URB_GET_FRAME_LENGTH UrbGetFrameLength;
    struct _URB_SET_FRAME_LENGTH UrbSetFrameLength;
    struct _URB_GET_CURRENT_FRAME_NUMBER UrbGetCurrentFrameNumber;
    struct _URB_CONTROL_TRANSFER UrbControlTransfer;
    struct _URB_BULK_OR_INTERRUPT_TRANSFER UrbBulkOrInterruptTransfer;
    struct _URB_ISOCH_TRANSFER UrbIsochronousTransfer;
    struct _URB_CONTROL_DESCRIPTOR_REQUEST UrbControlDescriptorRequest;
    struct _URB_CONTROL_GET_STATUS_REQUEST UrbControlGetStatusRequest;
    struct _URB_CONTROL_FEATURE_REQUEST UrbControlFeatureRequest;
    struct _URB_CONTROL_SYNC_FRAME_REQUEST UrbControlSyncFrameRequest;
    struct _URB_CONTROL_VENDOR_OR_CLASS_REQUEST UrbControlVendorClassRequest;
    struct _URB_CONTROL_GET_INTERFACE_REQUEST UrbControlGetInterfaceRequest;
    struct _URB_CONTROL_GET_CONFIGURATION_REQUEST UrbControlGetConfigurationRequest;
  }
} URB, *PURB ;
```

USB client drivers set up USB request blocks (URB) to send requests to the host controller driver. The URB structure defines a format for all possible commands that can be sent to a USB device.

Members

UrbHeader

Defines the format for requests that do not require additional structure data.

UrbSelectInterface

Defines the format of a select interface command for a USB device.

UrbSelectConfiguration

Defines the format of a select configuration command for a USB device.

UrbPipeRequest

Defines the format for a command to reset a stalled pipe on a USB device.

UrbFrameLengthControl

Defines the format for a command to take or release control of the frame length on a USB bus.

UrbGetFrameLength

Defines the format for a command to get the current frame length on a USB bus.

UrbSetFrameLength

Defines the format for a command to alter the frame length on a USB bus.

UrbGetCurrentFrameNumber

Defines the format for a command to get the current frame number on a USB bus.

UrbControlTransfer

Defines the format for a command to transmit or receive data on a control pipe.

UrbBulkOrInterruptTransfer

Defines the format for a command to transmit or receive data on a bulk pipe, or to receive data from an interrupt pipe.

UrbIsochronousTransfer

Defines the format of an isochronous transfer to a USB device.

UrbControlDescriptorRequest

Defines the format for a command to retrieve or set descriptor(s) on a USB device.

UrbControlGetStatusRequest

Defines the format for a command to get status from a device, interface, or endpoint.

UrbControlFeatureRequest

Defines the format for a command to set or clear USB-defined features on a device, interface, or endpoint.

UrbControlSyncFrameRequest

Defines the format for a command to get the frame number of an isochronous pattern transfer.

UrbControlVendorClassRequest

Defines the format for a command to send or receive a vendor or class-specific request on a device, interface, endpoint, or other device-defined target.

UrbControlGetInterfaceRequest

Defines the format for a command to get the current alternate interface setting for a selected interface.

UrbControlGetConfigurationRequest

Defines the format for a command to get the current configuration for a device.

Comments

For information on the function codes to set in each structure see _URB_HEADER.

See Also

_URB_HEADER, _URB_SELECT_INTERFACE, _URB_SELECT_CONFIGURATION, _URB_PIPE_REQUEST, _URB_FRAME_LENGTH_CONTROL, _URB_GET_FRAME_LENGTH, _URB_SET_FRAME_LENGTH, _URB_GET_CURRENT_FRAME_NUMBER, _URB_CONTROL_TRANSFER, _URB_BULK_OR_INTERRUPT_TRANSFER, _URB_ISOCH_TRANSFER, _URB_CONTROL_DESCRIPTOR_REQUEST, _URB_CONTROL_GET_STATUS_REQUEST, _URB_CONTROL_FEATURE_REQUEST, _URB_CONTROL_VENDOR_OR_CLASS_REQUEST, _URB_CONTROL_GET_INTERFACE_REQUEST, _URB_CONTROL_GET_CONFIGURATION_REQUEST

_URB_BULK_OR_INTERRUPT_TRANSFER

```
struct _URB_BULK_OR_INTERRUPT_TRANSFER {
  struct _URB_HEADER Hdr;
  USBD_PIPE_HANDLE PipeHandle;
  ULONG TransferFlags;
  ULONG TransferBufferLength;
  PVOID TransferBuffer;
  PMDL TransferBufferMDL;
  struct _URB *UrbLink;
  .
  .
};
```

USB client drivers set up this structure to transmit or receive data on a bulk pipe, or receive data on an interrupt pipe.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER, and **Hdr.Length** must be sizeof(_URB_BULK_OR_INTERRUPT_TRANSFER).

PipeHandle

Specifies the handle for this pipe returned by the host controller driver when a configuration was selected.

TransferFlags

Specifies zero, one, or a combination of the following flags:

USBD_TRANSFER_DIRECTION_IN

Is set to request data from a device. To transfer data to a device, this flag must be clear. The flag must be set if the pipe is an interrupt transfer pipe.

USBD_SHORT_TRANSFER_OK

Can be used if USBD_TRANSFER_DIRECTION_IN is set. If set, directs the host controller driver not to return an error if a packet is received from the device shorter than the maximum packet size for the endpoint. Otherwise, a short request is returns an error condition.

TransferBufferLength

Specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

TransferBuffer

Points to a resident buffer for the transfer or is NULL if an MDL is supplied in **TransferBufferMDL**. The contents of this buffer depend on the value of **TransferFlags**. If USBD_TRANSFER_DIRECTION_IN is specified this buffer will contain data read from the device on return from the host controller driver. Otherwise, this buffer contains driver-supplied data for transfer to the device.

TransferBufferMDL

Points to an MDL that describes a resident buffer or is NULL if a buffer is supplied in **TransferBuffer**. The contents of the buffer depend on the value of **TransferFlags**. If USBD_TRANSFER_DIRECTION_IN is specified, the described buffer will contain data read from the device on return from the host controller driver. Otherwise, the buffer contains driver-supplied data for transfer to the device. This MDL must be allocated from nonpaged pool.

UrbLink

Points to a caller-initialized URB. **UrbLink** becomes the subsequent URB in a chain of requests with this URB being the predecessor.

Comments

Drivers can use the **UsbBuildInterruptOrBulkTransferRequest** service routine to format this URB.Buffers specified in **TransferBuffer** or described in **TransferBufferMDL** must be nonpageable.

Other fields that are part of this structure, but not described here, should be treated as opaque and considered reserved for system use.

See Also

URB, _URB_HEADER

_URB_CONTROL_DESCRIPTOR_REQUEST

```
struct _URB_CONTROL_DESCRIPTOR_REQUEST {
  struct _URB_HEADER Hdr;
  .
  .
  ULONG TransferBufferLength ;
  PVOID TransferBuffer ;
  PMDL TransferBufferMDL ;
  struct _URB *UrbLink ;
  .
  .
  UCHAR Index ;
  UCHAR DescriptorType ;
  USHORT LanguageId ;
  .
  .
} ;
```

USB client drivers set up this structure to get or set descriptors on a USB device.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be one of URB_FUNCTION_GET_DESCRIPTOR_FROM_XXX or URB_FUNCTION_SET_DESCRIPTOR_FROM_XXX, and **Hdr.Length** must be sizeof(_URB_CONTROL_DESCRIPTOR_REQUEST).

TransferBufferLength

Specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

TransferBuffer

Points to a resident buffer for the transfer or is NULL if an MDL is supplied in **TransferBufferMDL**.

TransferBufferMDL

Points to an MDL that describes a resident buffer or is NULL if a buffer is supplied in **TransferBuffer**.This MDL must be allocated from nonpaged pool.

UrbLink

Points to an caller-initialized URB. **UrbLink** becomes the subsequent URB in a chain of requests with this URB being the predecessor.

Index

Specifies the device-defined index of the descriptor that is being retrieved or set.

DescriptorType

Indicates what type of descriptor is being retrieved or set. One of the following values must be specified:

USB_DEVICE_DESCRIPTOR_TYPE
USB_CONFIGURATION_DESCRIPTOR_TYPE
USB_STRING_DESCRIPTOR_TYPE

LanguageId

Specifies the language ID of the descriptor to be retrieved when USB_STRING_DESCRIPTOR_TYPE is set in **DescriptorType**. This member must be set to zero for any other value in **DescriptorType**.

Comments

Drivers can use the **UsbBuildGetDescriptorRequest** service routine to format this URB. If the caller passes a buffer too small to hold all of the data, the bus driver truncates the data to fit in the buffer without error.

When the caller requests the device descriptor, the bus driver returns a USB_DEVICE_DESCRIPTOR data structure.

When the caller requests a configuration descriptor, the bus driver returns the configuration descriptor in a USB_CONFIGURATION_DESCRIPTOR structure, followed by the interface and endpoint descriptors for that configuration. The driver can access the interface and endpoint descriptors as USB_INTERFACE_DESCRIPTOR, and USB_ENDPOINT_DESCRIPTOR structures. The bus driver also returns any class-specific or device-specific descriptors. The system provides the USBD_ParseConfigurationDescriptorEx and USBD_ParseDescriptors service routines to find individual descriptors within the buffer.

When the caller requests a string descriptor, the bus driver returns a USB_STRING_DESCRIPTOR structure. The string itself is found in the variable-length **bString** member of the string descriptor.

Other fields that are part of this structure, but not described here, should be treated as opaque and considered reserved for system use.

See Also

URB, _URB_HEADER

_URB_CONTROL_FEATURE_REQUEST

```
struct _URB_CONTROL_FEATURE_REQUEST {
  struct _URB_HEADER Hdr;
  .
  .
  struct _URB *UrbLink ;
  .
  .
  USHORT FeatureSelector ;
  USHORT Index ;
  .
  .
} ;
```

USB client drivers set up this structure to set or clear features on a device, interface, or endpoint.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be one of URB_FUNCTION_SET_FEATURE_TO_XXX or URB_FUNCTION_CLEAR_FEATURE_TO_XXX, and **Hdr.Length** must be sizeof(_URB_CONTROL_FEATURE_REQUEST).

UrbLink

Points to an caller-initialized URB. **UrbLink** becomes the subsequent URB in a chain of requests with this URB being the predecessor.

FeatureSelector

Is the USB-defined feature code to be cleared or set. Using a feature code that is invalid, cannot be set, or cannot be cleared will cause the target to stall.

Index

Specifies the device-defined index, returned by a successful configuration request, if the request is for an endpoint or interface. Otherwise, **Index** must be zero.

Comments

Drivers can use the **UsbBuildFeatureRequest** service routine to format this URB.

Other fields that are part of this structure but not described here should be treated as opaque and considered reserved for system use.

See Also

URB, _URB_HEADER

_URB_CONTROL_GET_CONFIGURATION_REQUEST

```
struct _URB_CONTROL_GET_CONFIGURATION_REQUEST {
  struct _URB_HEADER Hdr;
  .
  .
  ULONG TransferBufferLength;
  PVOID TransferBuffer;
  PMDL TransferBufferMDL;
  struct _URB *UrbLink;
  .
  .
} ;
```

USB client drivers set up this structure to retrieve the current configuration for a device.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_GET_CONFIGURATION, and **Hdr.Length** must be sizeof(_URB_CONTROL_GET_CONFIGURATION_REQUEST).

TransferBufferLength

Must be 1. This member specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**.

TransferBuffer

Points to a resident buffer for the transfer or is NULL if an MDL is supplied in **TransferBufferMDL**. The bus driver returns a single byte that specifies the index of the current configuration.

TransferBufferMDL

Points to an MDL that describes a resident buffer or is NULL if a buffer is supplied in **TransferBuffer**. The bus driver returns a single byte that specifies the index of the current configuration. This MDL must be allocated from nonpaged pool.

UrbLink

Points to an caller-initialized URB. **UrbLink** becomes the subsequent URB in a chain of requests with this URB being the predecessor.

Comments

Other fields that are part of this structure but not described here should be treated as opaque and considered reserved for system use.

See Also

URB, _URB_HEADER

_URB_CONTROL_GET_INTERFACE_REQUEST

```
struct _URB_CONTROL_GET_INTERFACE_REQUEST {
  struct _URB_HEADER Hdr;
  .
  .

 ULONG TransferBufferLength ;
  PVOID TransferBuffer ;
  PMDL TransferBufferMDL ;
  struct _URB *UrbLink ;
  .
  .
  USHORT Interface;
  .
  .
} ;
```

USB client drivers set up this structure to retrieve the current alternate interface setting for a interface in the current configuration.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_GET_INTERFACE, and **Hdr.Length** must be sizeof(_URB_CONTROL_GET_INTERFACE_REQUEST).

TransferBufferLength

Must be 1. This member specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

TransferBuffer

Points to a resident buffer for the transfer or is NULL if an MDL is supplied in **TransferBufferMDL**. The bus driver returns a single byte specifying the index of the current alternate setting for the interface.

TransferBufferMDL

Points to an MDL that describes a resident buffer or is NULL if a buffer is supplied in **TransferBuffer**. The bus driver returns a single byte specifying the index of the current alternate setting for the interface. This MDL must be allocated from nonpaged pool.

UrbLink

Points to an caller-initialized URB. **UrbLink** becomes the subsequent URB in a chain of requests with this URB being the predecessor.

Index

Specifies the device-defined index of the interface descriptor being retrieved.

Comments

Other fields that are part of this structure but not described here should be treated as opaque and considered reserved for system use.

See Also

URB, _URB_HEADER

_URB_CONTROL_GET_STATUS_REQUEST

```
struct _URB_CONTROL_GET_STATUS_REQUEST {
  struct _URB_HEADER Hdr;
  .
  .
  ULONG TransferBufferLength ;
  PVOID TransferBuffer ;
  PMDL TransferBufferMDL ;
  struct _URB *UrbLink ;
  .
  .
  USHORT Index ;
  .
  .
} ;
```

USB client drivers set up this structure to retrieve status from a device, interface, endpoint, or other device-defined target.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_ GET_STATUS, and **Hdr.Length** must be sizeof(_URB_CONTROL_GET_STATUS_ REQUEST).

TransferBufferLength

Must be 1. This member specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

TransferBuffer

Points to a resident buffer for the transfer or is NULL if an MDL is supplied in **TransferBufferMDL**. The bus driver returns a single byte specifying the index of the current alternate setting for the interface.

TransferBufferMDL

Points to an MDL that describes a resident buffer or is NULL if a buffer is supplied in **TransferBuffer**. The bus driver returns a single byte specifying the index of the current alternate setting for the interface. This MDL must be allocated from nonpaged pool.

UrbLink

Points to an caller-initialized URB. **UrbLink** becomes the subsequent URB in a chain of requests with this URB being the predecessor.

Index

Specifies the device-defined index, returned by a successful configuration request, if the request is for an endpoint or interface. Otherwise, **Index** must be zero.

Comments

Drivers can use the **UsbBuildGetStatusRequest** service routine to format this URB.

Other fields that are part of this structure but not described here should be treated as opaque and considered reserved for system use.

See Also

URB, _URB_HEADER

_URB_CONTROL_TRANSFER

```
struct _URB_CONTROL_TRANSFER {
  struct _URB_HEADER Hdr;
  USBD_PIPE_HANDLE PipeHandle;
  ULONG TransferFlags;
  ULONG TransferBufferLength;
  PVOID TransferBuffer;
  PMDL TransferBufferMDL;
```

```
  struct _URB *UrbLink;
  .
  .
  UCHAR SetupPacket[8];
} ;
```

USB client drivers set up this structure to transfer data to or from a control pipe.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_CONTROL_TRANSFER, and **Hdr.Length** must be sizeof(_URB_CONTROL_TRANSFER).

PipeHandle

Specifies the handle for this pipe returned by the host controller driver when a configuration was selected.

TransferFlags

Specifies zero, one, or a combination of the following flags:

USBD_TRANSFER_DIRECTION_IN

Is set to request data from a device. To transfer data to a device, this flag must be clear. The flag must be set if the pipe is an interrupt transfer pipe.

USBD_SHORT_TRANSFER_OK

Can be used if USBD_TRANSFER_DIRECTION_IN is set. If set, directs the host controller driver not to return an error if a packet is received from the device shorter than the maximum packet size for the endpoint. Otherwise, a short request is returns an error condition.

TransferBufferLength

Specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

TransferBuffer

Points to a resident buffer for the transfer or is NULL if an MDL is supplied in **TransferBufferMDL**. The contents of this buffer depend on the value of **TransferFlags**. If USBD_TRANSFER_DIRECTION_IN is specified this buffer will contain data read from the device on return from the host controller driver. Otherwise, this buffer contains driver-supplied data for transfer to the device.

TransferBufferMDL

Points to an MDL that describes a resident buffer or is NULL if a buffer is supplied in **TransferBuffer**. The contents of the buffer depend on the value of **TransferFlags**. If USBD_TRANSFER_DIRECTION_IN is specified, the described buffer will contain data read from the device on return from the host controller driver. Otherwise, the buffer contains driver-supplied data for transfer to the device. This MDL must be allocated from nonpaged pool.

UrbLink

Points to an caller-initialized URB. **UrbLink** becomes the subsequent URB in a chain of requests with this URB being the predecessor.

SetupPacket

Is a USB-defined request setup packet. The format of a USB request setup packet is found in the USB core specification.

Comments

Other fields that are part of this structure but not described here should be treated as opaque and considered reserved for system use.

See Also

URB, _URB_HEADER

_URB_CONTROL_VENDOR_OR_CLASS_REQUEST

```
struct _URB_CONTROL_VENDOR_OR_CLASS_REQUEST {
  struct _URB_HEADER Hdr;
  .
  .
  ULONG TransferFlags ;
  ULONG TransferBufferLength ;
  PVOID TransferBuffer ;
  PMDL TransferBufferMDL ;
  struct _URB *UrbLink ;
  .
  .
  UCHAR RequestTypeReservedBits;
  UCHAR Request;
  USHORT Value;
  USHORT Index;
  .
  .
};
```

USB client drivers set up this structure to issue a vendor or class-specific command to a device, interface, endpoint, or other device-defined target.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be one of URB_FUNCTION_CLASS_XXX or URB_FUNCTION_VENDOR_XXX GET_STATUS, and **Hdr.Length** must be sizeof(_URB_CONTROL_VENDOR_OR_CLASS_REQUEST).

TransferFlags

Specifies zero, one, or a combination of the following flags:

USBD_TRANSFER_DIRECTION_IN

Is set to request data from a device. To transfer data to a device, this flag must be clear. The flag must be set if the pipe is an interrupt transfer pipe.

USBD_SHORT_TRANSFER_OK

Can be used if USBD_TRANSFER_DIRECTION_IN is set. If set, directs the host controller driver not to return an error if a packet is received from the device shorter than the maximum packet size for the endpoint. Otherwise, a short request is returns an error condition.

TransferBufferLength

Specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

TransferBuffer

Points to a resident buffer for the transfer or is NULL if an MDL is supplied in **TransferBufferMDL**. The contents of this buffer depend on the value of **TransferFlags**. If USBD_TRANSFER_DIRECTION_IN is specified this buffer will contain data read from the device on return from the host controller driver. Otherwise, this buffer contains driver-supplied data for transfer to the device.

TransferBufferMDL

Points to an MDL that describes a resident buffer or is NULL if a buffer is supplied in **TransferBuffer**. The contents of the buffer depend on the value of **TransferFlags**. If USBD_TRANSFER_DIRECTION_IN is specified, the described buffer will contain data read from the device on return from the host controller driver. Otherwise, the buffer contains driver-supplied data for transfer to the device. This MDL must be allocated from non-paged pool.

UrbLink

Points to a caller-initialized URB. **UrbLink** becomes the subsequent URB in a chain of requests with this URB being the predecessor.

RequestTypeReservedBits

Specifies a value, from 4 to 31 inclusive, that becomes part of the request type code in the USB-defined setup packet. This value is defined by USB for a class request or the vendor for a vendor request.

Request

Specifies the USB or vendor-defined request code for the device, interface, endpoint, or other device-defined target.

Value

Is a value, specific to **Request**, that becomes part of the USB-defined setup packet for the target. This value is defined by the creator of the code used in **Request**.

Index

Specifies the device-defined index, returned by a successful configuration request, if the request is for an endpoint or interface. Otherwise, **Index** must be zero.

Comments

Drivers can use the **UsbBuildVendorRequest** service routine format this URB.

Other fields that are part of this structure but not described here should be treated as opaque and considered to be reserved for system use.

See Also

URB, _URB_HEADER

_URB_FRAME_LENGTH_CONTROL

```
struct _URB_FRAME_LENGTH_CONTROL {
  struct _URB_HEADER Hdr;
} ;
```

USB client drivers set up this structure to take or release control of the frame length on the bus.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be either URB_FUNCTION_TAKE_FRAME_LENGTH_CONTROL or URB_FUNCTION_RELEASE_FRAME_LENGTH_CONTROL, and **Hdr.Length** must be sizeof(_URB_FRAME_LENGTH_CONTROL).

Comments

Only one client can have control of frame length at any time. To take control of the frame length, a driver sets **Hdr.Function** to URB_FUNCTION_TAKE_FRAME_LENGTH_CONTROL. When a client, who has taken control of the frame length, is ready to release control, it must release control it by sending this URB with **Hdr.Function** set to URB_FUNCTION_RELEASE_FRAME_LENGTH_CONTROL.

See Also

URB, _URB_HEADER

_URB_GET_CURRENT_FRAME_NUMBER

```
struct _URB_GET_CURRENT_FRAME_NUMBER {
  struct _URB_HEADER Hdr;
  ULONG FrameNumber ;
} ;
```

USB client drivers set up this structure to retrieve the current frame length on the bus.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_GET_FRAME_LENGTH, and **Hdr.Length** must be sizeof(_URB_GET_CURRENT_FRAME_NUMBER).

FrameNumber

Contains the current frame number, on the USB bus, on return from the host controller driver.

See Also

URB, _URB_HEADER

_URB_GET_FRAME_LENGTH

```
struct _URB_GET_FRAME_LENGTH {
  struct _URB_HEADER Hdr;
  ULONG FrameLength ;
  ULONG FrameNumber ;
} ;
```

USB client drivers set up this structure to retrieve the current frame length on the bus.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_GET_FRAME_LENGTH, and **Hdr.Length** must be sizeof(_URB_GET_FRAME_LENGTH).

FrameLength

Contains the length of each bus frame in USB-defined bit times.

FrameNumber

Contains the earliest bus frame number that the frame length can be altered on return from the host controller driver.

Comments

Client drivers must request and be granted sole control of frame length on the bus before submitting an URB of this type. Clients can take control of frame length settings by submitting an URB using _URB_FRAME_LENGTH_CONTROL as the data structure.

See Also

URB, _URB_HEADER, _URB_FRAME_LENGTH_CONTROL

_URB_HEADER

```
struct _URB_HEADER {
  USHORT Length ;
  USHORT Function ;
  .
  .
  USBD_STATUS Status ;
  .
  .
} ;
```

USB client drivers set up this structure to provide basic information about the request being sent to the host controller driver.

Members

Length

Specifies the length, in bytes, of the URB. For URB requests that use data structures other than _URB_HEADER, this member must be set to the length of the entire URB request structure, not the _URB_HEADER size.

Function

Specifies a numeric code indicating the requested operation for this URB. One of the following value must be set:

URB_FUNCTION_SELECT_CONFIGURATION

Indicates to the host controller driver that a configuration is to be selected. If set, the URB is used with _URB_SELECT_CONFIGURATION as the data structure.

URB_FUNCTION_SELECT_INTERFACE

Indicates to the host controller driver that an alternate interface setting is being selected for an interface. If set, the URB is used with _URB_SELECT_INTERFACE as the data structure.

URB_FUNCTION_ABORT_PIPE

Indicates that all outstanding requests for a pipe should be canceled. If set, the URB is used with _URB_PIPE_REQUEST as the data structure.

URB_FUNCTION_TAKE_FRAME_LENGTH_CONTROL

Indicates that the client is requesting sole control of the frame length for the USB bus. If set, the URB is used with _URB_FRAME_LENGTH_CONTROL as the data structure.

URB_FUNCTION_RELEASE_FRAME_LENGTH_CONTROL

Indicates that the client is releasing control of the frame length on the USB bus. If set, the URB is used with _URB_FRAME_LENGTH_CONTROL as the data structure.

URB_FUNCTION_GET_FRAME_LENGTH

Requests the current frame length on the USB bus. If set, the URB is used with _URB_GET_FRAME_LENGTH as the data structure.

URB_FUNCTION_SET_FRAME_LENGTH

Alters the current frame length on the USB bus. If set, the URB is used with _URB_SET_FRAME_LENGTH as the data structure.

URB_FUNCTION_GET_CURRENT_FRAME_NUMBER

Requests the current frame number from the host controller driver. If set, the URB is used with _URB_GET_CURRENT_FRAME_NUMBER as the data structure.

URB_FUNCTION_CONTROL_TRANSFER

Transfers data to or from a control pipe. If set, the URB is used with _URB_CONTROL_TRANSFER as the data structure.

URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER

Transfers data from a bulk pipe or interrupt pipe or to an bulk pipe. If set, the URB is used with _URB_BULK_OR_INTERRUPT_TRANSFER as the data structure.

URB_FUNCTION_ISOCH_TRANSFER

Transfers data to or from an isochronous pipe. If set, the URB is used with _URB_ISOCH_TRANSFER as the data structure.

URB_FUNCTION_RESET_PIPE

Causes a stall condition on an endpoint to be cleared. If set, the URB is used with _URB_PIPE_REQUEST as the data structure.

URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE

Retrieves the device descriptor from a specific USB device. If set, the URB is used with _URB_CONTROL_DESCRIPTOR_REQUEST as the data structure.

URB_FUNCTION_GET_DESCRIPTOR_FROM_ENDPOINT

Retrieves the descriptor from an endpoint on an interface for a USB device. If set, the URB is used with _URB_CONTROL_DESCRIPTOR_REQUEST as the data structure.

URB_FUNCTION_SET_DESCRIPTOR_TO_DEVICE

Sets a device descriptor on a device. If set, the URB is used with _URB_CONTROL_DESCRIPTOR_REQUEST as the data structure.

URB_FUNCTION_SET_DESCRIPTOR_TO_ENDPOINT

Sets an endpoint descriptor on an endpoint for an interface. If set, the URB is used with _URB_CONTROL_DESCRIPTOR_REQUEST as the data structure.

URB_FUNCTION_SET_FEATURE_TO_DEVICE

Sets a USB-defined feature on a device. If set, the URB is used with _URB_CONTROL_FEATURE_REQUEST as the data structure.

URB_FUNCTION_SET_FEATURE_TO_INTERFACE

Sets a USB-defined feature on an interface for a device. If set, the URB is used with _URB_CONTROL_FEATURE_REQUEST as the data structure.

URB_FUNCTION_SET_FEATURE_TO_ENDPOINT

Sets a USB-defined feature on an endpoint for an interface on a USB device. If set, the URB is used with _URB_CONTROL_FEATURE_REQUEST as the data structure.

URB_FUNCTION_SET_FEATURE_TO_OTHER

Sets a USB-defined feature on a device-defined target on a USB device. If set, the URB is used with _URB_CONTROL_FEATURE_REQUEST as the data structure.

URB_FUNCTION_CLEAR_FEATURE_TO_DEVICE

Clears a USB-defined feature on a device. If set, the URB is used with _URB_CONTROL_FEATURE_REQUEST as the data structure.

URB_FUNCTION_CLEAR_FEATURE_TO_INTERFACE

Clears a USB-defined feature on an interface for a device. If set, the URB is used with _URB_CONTROL_FEATURE_REQUEST as the data structure.

URB_FUNCTION_CLEAR_FEATURE_TO_ENDPOINT

Clears a USB-defined feature on an endpoint, for an interface, on a USB device. If set, the URB is used with _URB_CONTROL_FEATURE_REQUEST as the data structure.

URB_FUNCTION_CLEAR_FEATURE_TO_OTHER

Clears a USB-defined feature on a device defined target on a USB device. If set, the URB is used with _URB_CONTROL_FEATURE_REQUEST as the data structure.

URB_FUNCTION_GET_STATUS_FROM_DEVICE

Retrieves status from a USB device. If set, the URB is used with _URB_CONTROL_GET_STATUS_REQUEST as the data structure.

URB_FUNCTION_GET_STATUS_FROM_INTERFACE

Retrieves status from an interface on a USB device. If set, the URB is used with _URB_CONTROL_GET_STATUS_REQUEST as the data structure.

URB_FUNCTION_GET_STATUS_FROM_ENDPOINT

Retrieves status from an endpoint for an interface on a USB device. If set, the URB is used with _URB_CONTROL_GET_STATUS_REQUEST as the data structure.

URB_FUNCTION_GET_STATUS_FROM_OTHER

Retrieves status from a device-defined target on a USB device. If set, the URB is used with _URB_CONTROL_GET_STATUS_REQUEST as the data structure.

URB_FUNCTION_VENDOR_DEVICE

Sends a vendor-specific command to a USB device. If set, the URB is used with _URB_CONTROL_VENDOR_OR_CLASS_REQUEST as the data structure.

URB_FUNCTION_VENDOR_INTERFACE

Sends a vendor-specific command for an interface on a USB device. If set, the URB is used with _URB_CONTROL_VENDOR_OR_CLASS_REQUEST as the data structure.

URB_FUNCTION_VENDOR_ENDPOINT

Sends a vendor-specific command for an endpoint on an interface on a USB device. If set, the URB is used with _URB_CONTROL_VENDOR_OR_CLASS_REQUEST as the data structure.

URB_FUNCTION_VENDOR_OTHER

Sends a vendor-specific command to a device-defined target on a USB device. If set, the URB is used with _URB_CONTROL_VENDOR_OR_CLASS_REQUEST as the data structure.

URB_FUNCTION_CLASS_DEVICE

Sends a USB-defined class-specific command to a USB device. If set, the URB is used with _URB_CONTROL_VENDOR_OR_CLASS_REQUEST as the data structure.

URB_FUNCTION_CLASS_INTERFACE

Sends a USB-defined class-specific command to an interface on a USB device. If set, the URB is used with _URB_CONTROL_VENDOR_OR_CLASS_REQUEST as the data structure.

URB_FUNCTION_CLASS_ENDPOINT

Sends a USB-defined class-specific command to an endpoint, on an interface, on a USB device. If set, the URB is used with _URB_CONTROL_VENDOR_OR_CLASS_ REQUEST as the data structure.

URB_FUNCTION_CLASS_OTHER

Sends a USB-defined class-specific command to a device defined target on a USB device. If set, the URB is used with _URB_CONTROL_VENDOR_OR_CLASS_REQUEST as the data structure.

URB_FUNCTION_GET_CONFIGURATION

Retrieves the current configuration on a USB device. If set, the URB is used with _URB_ CONTROL_GET_CONFIGURATION_REQUEST as the data structure.

URB_FUNCTION_GET_INTERFACE

Retrieves the current settings for an interface on a USB device. If set, the URB is used with _URB_CONTROL_GET_INTERFACE_REQUEST as the data structure.

Status

Contains a USB_STATUS_XXX code on return from the host controller driver.

Comments

The _URB_HEADER structure is a member of all USB requests that are part of the URB structure. The _URB_HEADER structure is used to provide common information about each request to the host controller driver.

Other fields that are part of this structure but not described here should be treated as opaque and considered to be reserved for system use.

See Also

URB, _URB_SELECT_INTERFACE, _URB_SELECT_CONFIGURATION, _URB_PIPE_REQUEST, _URB_FRAME_LENGTH_CONTROL, _URB_GET_FRAME_LENGTH, _URB_SET_FRAME_LENGTH, _URB_GET_CURRENT_FRAME_NUMBER, _URB_CONTROL_TRANSFER, _URB_BULK_OR_INTERRUPT_TRANSFER, _URB_ISOCH_TRANSFER, _URB_CONTROL_DESCRIPTOR_REQUEST, _URB_CONTROL_GET_STATUS_REQUEST, _URB_CONTROL_FEATURE_REQUEST, _URB_CONTROL_VENDOR_OR_CLASS_REQUEST, _URB_CONTROL_GET_INTERFACE_REQUEST, _URB_CONTROL_GET_CONFIGURATION_REQUEST

_URB_ISOCH_TRANSFER

```
struct _URB_ISOCH_TRANSFER {
  struct _URB_HEADER Hdr;
  USBD_PIPE_HANDLE PipeHandle;
  ULONG TransferFlags;
  ULONG TransferBufferLength;
  PVOID TransferBuffer;
  PMDL TransferBufferMDL;
  .
  .
  ULONG StartFrame;
  ULONG NumberOfPackets;
  ULONG ErrorCount;
  USBD_ISO_PACKET_DESCRIPTOR IsoPacket[1];
};
```

USB client drivers set up this structure to send data to or retrieve data from an isochronous transfer pipe.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_ISOCH_TRANSFER, and **Hdr.Length** must be the size of this variable-length data structure.

PipeHandle

Specifies the handle for this pipe returned by the host controller driver when a configuration was selected.

TransferFlags

Specifies zero, one, or a combination of the following flags:

USBD_TRANSFER_DIRECTION_IN

Is set to request data from a device. To transfer data to a device, this flag must be clear. The flag must be set if the pipe is an interrupt transfer pipe.

USBD_SHORT_TRANSFER_OK

Can be used if USBD_TRANSFER_DIRECTION_IN is set. If set, directs the host controller driver not to return an error if a packet is received from the device shorter than the maximum packet size for the endpoint. Otherwise, a short request is returns an error condition.

USBD_ISO_TRANSFER_ASAP

Causes the transfer to begin on the next frame, if no transfers have been submitted to the pipe since the pipe was opened or last reset. Otherwise, the transfer will begin on the first frame following all currently queued requests for the pipe. The actual frame that the transfer begins on will be adjusted for bus latency by the host controller driver.

TransferBufferLength

Specifies the length, in bytes, of the buffer specified in **TransferBuffer** or described in **TransferBufferMDL**. The host controller driver returns the number of bytes sent to or read from the pipe in this member.

TransferBuffer

Points to a resident buffer for the transfer or is NULL if an MDL is supplied in **TransferBufferMDL**. The contents of this buffer depend on the value of **TransferFlags**. If USBD_TRANSFER_DIRECTION_IN is specified this buffer will contain data read from the device on return from the host controller driver. Otherwise, this buffer contains driver-supplied data for transfer to the device.

TransferBufferMDL

Points to an MDL that describes a resident buffer or is NULL if a buffer is supplied in **TransferBuffer**. The contents of the buffer depend on the value of **TransferFlags**. If USBD_TRANSFER_DIRECTION_IN is specified, the described buffer will contain data read from the device on return from the host controller driver. Otherwise, the buffer contains driver-supplied data for transfer to the device. This MDL must be allocated from non-paged pool.

StartFrame

Specifies the frame number the transfer should begin on. This variable must be within a system-defined range of the current frame. The range is specified by the constant USBD_ISO_START_FRAME_RANGE.

If START_ISO_TRANSFER_ASAP is set in **TransferFlags**, this member contains the frame number that the transfer began on, when the request is returned by the host controller driver. Otherwise, this member must contain the frame number that this transfer will begin on.

NumberOfPackets

Specifies the number of packets described by the boundless array member **IsoPacket**.

ErrorCount

Contains the number of packets that completed with an error condition on return from the host controller driver.

IsoPacket

Contains a variable-length array of USBD_ISO_PACKET_DESCRIPTOR structures that describe each transfer packet of the isochronous transfer.

Comments

Each entry in the **IsoPacket** member array specifies an offset and a length within the transfer buffer for the request. If IsoPacket has *n* entries, the host controller transfers use *n* frames to transfer data, transferring **IsoPacket[*i*].Length** bytes beginning at an offset of **IsoPacket[*i*].Offset**.

Drivers can use the GET_ISO_URB_SIZE macro to determine the size needed to hold the entire URB.

Other fields that are part of this structure but not described here should be treated as opaque and considered to be reserved for system use.

See Also

URB, _URB_HEADER, USBD_ISO_PACKET_DESCRIPTOR

_URB_PIPE_REQUEST

```
struct _URB_PIPE_REQUEST {
  struct _URB_HEADER Hdr;
  USBD_PIPE_HANDLE PipeHandle ;
} ;
```

USB client drivers set up this structure to clear a stall condition on an endpoint.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_RESET_PIPE or URB_FUNCTION_ABORT_PIPE, and **Hdr.Length** must be sizeof(_URB_PIPE_REQUEST).

PipeHandle

Specifies the handle for this pipe returned by the host controller driver when a configuration was selected.

Comments

Other fields that are part of this structure but not described here should be treated as opaque and considered to be reserved for system use.

See Also

URB, _URB_HEADER

_URB_SELECT_CONFIGURATION

```
struct _USB_SELECT_CONFIGURATION {
  struct _URB_HEADER Hdr;
  PUSB_CONFIGRUATION_DESCRIPTOR ConfigurationDescriptor ;
  USB_CONFIGURATION_HANDLE ConfigurationHandle ;
  USBD_INTERFACE_INFORMATION Interface ;
} ;
```

USB client drivers set up this structure to select a configuration for a USB device.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_SELECT_CONFIGURATION, and **Hdr.Length** must be the size of the entire URB.

ConfigurationDescriptor

Points to an initialized USB configuration descriptor that identifies the configuration to be used on the device. If this member is NULL, the device will be set into an unconfigured state.

ConfigurationHandle

Contains a handle that is used to access this configuration on return from the host controller driver. USB client drivers must treat this member as opaque.

Interface

Specifies a variable length array of USBD_INTERFACE_INFORMATION structures, each describing an interface supported by the configuration being selected.

Before the request is sent to the host controller driver, the driver may select an alternate setting for one or more of the interfaces contained in this array by setting members of the USBD_INTERFACE_INFORMATION structure for that interface.

On return from the host controller driver, this member contains a USBD_INTERFACE_INFORMATION structure with data about the capabilities and format of the endpoints within that interface.

Comments

An URB_FUNCTION_SELECT_CONFIGURATION URB consists of a _URB_SELECT_CONFIGURATION structure followed by a sequence of variable-length USB_INTERFACE_INFORMATION structures. Drivers can use the USBD_CreateConfigurationRequestEx service routine to allocate the URB.

Other fields that are part of this structure but not described here should be treated as opaque and considered to be reserved for system use.

See Also

URB, _URB_HEADER, USBD_INTERFACE_INFORMATION

_URB_SELECT_INTERFACE

```
struct _URB_SELECT_INTERFACE {
  struct _URB_HEADER Hdr;
  USBD_CONFIGURATION_HANDLE ConfigurationHandle ;
  USBD_INTERFACE_INFORMATION Interface ;
} ;
```

USB client drivers set up this structure to select an alternate setting for a interface in the current configuration on a USB device.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_SELECT_INTERFACE, and **Hdr.Length** must be the size of the entire URB.

ConfigurationHandle

Specifies the handle for the configuration that this interface belongs to, returned by the host controller driver when a configuration was selected.

Interface

A variable-length structure that specifies the interface and the new alternate setting for that interface. See USBD_INTERFACE_INFORMATION for information on the members that are used to control those settings. On successful completion of processing this URB, the bus driver returns an array of handles for each pipe on this interface in the **Interface.Pipes** array member.

Comments

Drivers can use the GET_SELECT_INTERFACE_REQUEST_SIZE macro to determine the size of this URB, and the **UsbBuildSelectInterfaceRequest** routine to format the URB.

Other fields that are part of this structure but not described here should be treated as opaque and considered to be reserved for system use.

See Also

URB, _URB_HEADER, USBD_INTERFACE_INFORMATION

_URB_SET_FRAME_LENGTH

```
struct _URB_SET_FRAME_LENGTH {
  struct _URB_HEADER Hdr;
  LONG FrameLengthDelta ;
} ;
```

USB client drivers set up this structure to change the frame length on the bus.

Members

Hdr

Specifies the URB header information. **Hdr.Function** must be URB_FUNCTION_SET_FRAME_LENGTH, and **Hdr.Length** must be sizeof(_URB_SET_FRAME_LENGTH).

FrameLengthDelta

Specifies the number of USB-defined bit times to be added or subtracted from the current frame length. The maximum increase or decrease per URB is one.

Comments

Sending this request to the host controller driver before being granted control of frame length for the bus will cause an error to be returned. To take control of the frame length, an URB must first be sent with the function, URB_FUNCTION_TAKE_FRAME_LENGTH_CONTROL. See _URB_FRAME_LENGTH_CONTROL for the data structure used to obtain or release control of frame length on the bus.

See Also

URB, _URB_HEADER, _URB_FRAME_LENGTH_CONTROL

USB_CONFIGURATION_DESCRIPTOR

```
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
  UCHAR bLength ;
  UCHAR bDescriptorType ;
  USHORT wTotalLength ;
  UCHAR bNumInterfaces ;
  .
  .
  UCHAR iConfiguration ;
  UCHAR bmAttributes ;
  UCHAR MaxPower ;
} USB_CONFIGURATION_DESCRIPTOR, *PUSB_CONFIGURATION_DESCRIPTOR ;
```

USB client drivers use this structure to hold a USB-defined configuration descriptor.

Members

bLength

Specifies the length, in bytes, of this structure.

bDescriptorType

Must be set to USB_CONFIGURATION_DESCRIPTOR_TYPE.

wTotalLength

Specifies the total length, in bytes, of all data for the configuration. The length includes all interface, endpoint, class, or vendor-specific descriptors returned with the configuration descriptor.

bNumInterfaces

Specifies the total number of interfaces supported by this configuration.

iConfiguration

Specifies the device-defined index of the string descriptor for this configuration.

bmAttributes

Specifies a bitmap to describe behavior of this configuration. The bits are described and set in little-endian order.

Bit	Meaning
0 - 4	Reserved.
5	Is set if this configuration supports remote wakeup.
6	Is set if this configuration is self-powered and does not use power from the bus.
7	Is set if this configuration is powered by the bus.

MaxPower

Specifies the power requirements of this device in two mA units. This field is valid only if bit seven is set in **bmAttributes**.

Comments

If **wTotalLength** is greater than the buffer size provided in the URB to hold all descriptors (interface, endpoint, class, and vendor-defined) retrieved, incomplete data will be returned. In order to retrieve complete descriptors, the request will need to be re-sent with a larger buffer.

If **bmAttributes** bits six and seven are both set, then the device is powered both by the bus and a source external to the bus.

Other fields that are part of this structure but not described here should be treated as opaque and considered to be reserved for system use.

See Also

UsbBuildGetDescriptorRequest, *USBD_CreateConfigurationRequest*

USB_DEVICE_DESCRIPTOR

```
typedef struct _USB_DEVICE_DESCRIPTOR {
  UCHAR bLength ;
  UCHAR bDescriptorType ;
  USHORT bcdUSB ;
  UCHAR bDeviceClass ;
  UCHAR bDeviceSubClass ;
  UCHAR bDeviceProtocol ;
  UCHAR bMaxPacketSize0 ;
  USHORT idVendor ;
```

```
  USHORT idProduct ;
  USHORT bcdDevice ;
  UCHAR iManufacturer ;
  UCHAR iProduct ;
  UCHAR iSerialNumber ;
  UCHAR bNumConfigurations ;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR ;
```

This structure is used by USB client drivers to hold a USB-defined device descriptor.

Members

bLength

Specifies the length, in bytes, of this descriptor.

bDescriptorType

Must be set to USB_DEVICE_DESCRIPTOR_TYPE.

bcdUSB

Identifies the version of the USB specification that this descriptor structure complies with. This value is a binary-coded decimal number.

bDeviceClass

Is the class code of the device as assigned by the USB specification group.

bDeviceSubClass

Is the subclass code of the device as assigned by the USB specification group.

bDeviceProtocol

Is the protocol code of the device as assigned by the USB specification group.

bMaxPacketSize0

Specifies the maximum packet size, in bytes, for endpoint zero of the device. The value must be set to 8, 16, 32, or 64.

idVendor

Is the vendor identifier for the device as assigned by the USB specification committee.

idProduct

Is the product identifier. This value is assigned by the manufacturer and is device-specific.

bcdDevice

Identifies the version of the device. This value is a binary-coded decimal number.

iManufacturer

Specifies a device-defined index of the string descriptor that provides a string containing the name of the manufacturer of this device.

iProduct

Specifies a device-defined index of the string descriptor that provides a string that contains a description of the device.

iSerialNumber

Specifies a device-defined index of the string descriptor that provides a string that contains a manufacturer-determined serial number for the device.

bNumConfigurations

Specifies the total number of possible configurations for the device.

Comments

This structure is used to hold a retrieved USB-defined device descriptor. This information can then be used to further configure or retrieve information about the device. Device descriptors are retrieved by submitting a get descriptor URB.

The **iManufacturer**, **iProduct**, and **iSerialNumber** values when returned from the host controller driver contain index values into an array of string descriptors maintained by the device. To retrieve these strings a string descriptor request can be sent to the device using these index values.

See Also

UsbBuildGetDescriptorRequest, _URB_CONTROL_DESCRIPTOR_REQUEST

USB_ENDPOINT_DESCRIPTOR

```
typedef struct _USB_ENDPOINT_DESCRIPTOR {
  UCHAR bLength ;
  UCHAR bDescriptorType ;
  UCHAR bEndpointAddress ;
  UCHAR bmAttributes ;
  USHORT wMaxPacketSize ;
  UCHAR bInterval ;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR ;
```

This structure is used by USB client drivers to hold a USB-defined endpoint descriptor.

Members

bLength

Specifies the length, in bytes, of this descriptor.

bDescriptorType

Must be set to USB_ENDPOINT_DESCRIPTOR_TYPE.

bEndpointAddress

Specifies the USB-defined endpoint address. The four low-order bits specify the endpoint number. The high-order bit specifies the direction of data flow on this endpoint: 1 for in, 0 for out.

bmAttributes

The two low-order bits specify the endpoint type, one of USB_ENDPOINT_TYPE_CONTROL, USB_ENDPOINT_TYPE_ISOCHRONOUS, USB_ENDPOINT_TYPE_BULK, USB_ENDPOINT_TYPE_INTERRUPT.

wMaxPacketSize

Specifies the maximum packet size that can be sent from or to this endpoint.

bInterval

For interrupt endpoints, **bInterval** specifies the polling interval, in frames.

USB_INTERFACE_DESCRIPTOR

```
typedef struct _USB_INTERFACE_DESCRIPTOR {
  UCHAR bLength ;
  UCHAR bDescriptorType ;
  UCHAR bInterfaceNumber ;
  UCHAR bAlternateSetting ;
  UCHAR bNumEndpoints ;
  UCHAR bInterfaceClass ;
  UCHAR bInterfaceSubClass ;
  UCHAR bInterfaceProtocol ;
  UCHAR iInterface ;
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR ;
```

This structure is used by USB client drivers to hold a USB-defined interface descriptor.

Members

bLength

Specifies the length, in bytes, of this descriptor.

bDescriptorType

Specifies the descriptor type. **bDescriptor** must be set to USB_INTERFACE_ DESCRIPTOR_TYPE.

bInterfaceNumber

Specifies the index number of this interface.

bAlternateSetting

Specifies the index number of this alternate setting of the interface.

bNumEndpoints

Specifies the number of endpoints that are used by the interface, excluding the default status endpoint.

bInterfaceClass

Is the class code of the device as assigned by the USB specification group.

bInterfaceSubClass

Is the subclass code of the device as assigned by the USB specification group.

bInterfaceProtocol

Is the protocol code of the device as assigned by the USB specification group.

iInterface

Specifies the index of a string descriptor that describes the interface. **iInterface** must be set to 0x1.

USB_HUB_NAME

```
typedef struct _USB_HUB_NAME {
  ULONG ActualLength;
  WCHAR HubName[1];
} USB_HUB_NAME, *PUSB_HUB_NAME;
```

This structure stores the hub's symbolic device name.

Members

ActualLength

Size of the entire data structure in bytes.

HubName

A pointer to this field will point to the Unicode string containing the hub's symbolic device name.

See Also

IOCTL_INTERNAL_USB_GET_CONTROLLER_NAME

USB_ROOT_HUB_NAME

```
typedef struct _USB_ROOT_HUB_NAME {
  ULONG ActualLength;
  WCHAR RootHubName[1];
} USB_ROOT_HUB_NAME, *PUSB_ROOT_HUB_NAME;
```

This structure stores the root hub's symbolic device name.

Members

ActualLength

Size of the entire data structure in bytes.

RootHubName

A pointer to this field will point to the Unicode string containing the root hub's symbolic device name.

See Also

IOCTL_INTERNAL_USB_GET_HUB_NAME

USB_STRING_DESCRIPTOR

```
typedef struct _USB_STRING_DESCRIPTOR {
  UCHAR bLength ;
  UCHAR bDescriptorType ;
  WCHAR bString[1] ;
} USB_STRING_DESCRIPTOR, *PUSB_STRING_DESCRIPTOR ;
```

This structure is used by USB client drivers to hold a USB-defined string descriptor.

Members

bLength

Specifies the length, in bytes, of the descriptor.

bDescriptorType

Must always be USB_STRING_DESCRIPTOR_TYPE.

bString

Points to a client-allocated buffer that contains, on return from the host controller driver, a Unicode string with the requested string descriptor.

Comments

This structure is used to hold a device, configuration, interface, class, vendor, endpoint, or device string descriptor. The string descriptor provides a human-readable description of the component.

Strings returned in **bString** are in Unicode format and the contents of the strings are device-defined.

See Also

UsbBuildGetDescriptorRequest, _URB_CONTROL_DESCRIPTOR_REQUEST

USBD_INTERFACE_INFORMATION

```
typedef struct _USBD_INTERFACE_INFORMATION {
  USHORT Length ;
  UCHAR InterfaceNumber ;
  UCHAR AlternateSetting ;
  UCHAR Class ;
  UCHAR SubClass ;
  UCHAR Protocol ;
  .
  .
  USBD_INTERFACE_HANDLE InterfaceHandle ;
  ULONG NumberOfPipes ;
  USBD_PIPE_INFORMATION Pipes[1] ;
} USBD_INTERFACE_INFORMATION, *PUSBD_INTERFACE_INFORMATION;
```

USB client drivers use this structure to hold information about an interface for a configuration on a USB device.

Members

Length

Specifies the length, in bytes, of this structure.

InterfaceNumber

Is the device-defined index identifier for this interface.

AlternateSetting

Specifies a device-defined index identifier that indicates which alternate setting this interface is using, should use, or describes.

Class

Is a USB-assigned identifier to specify a USB-defined class that this interface conforms to.

SubClass

Is a USB-assigned identifier to specify a USB-defined subclass that this interface conforms to. This code is specific to the code in **Class**.

Protocol

Is a USB-assigned identifier to specify a USB-defined protocol that this interface conforms to. This code is specific to the codes in **Class** and **SubClass**.

InterfaceHandle

Is a host controller driver-defined handle that is used to access this interface. This field should be treated as opaque.

NumberOfPipes

Specifies the number of pipes (endpoints) in this interface.

PipeInformation

Is a variable length array of USBD_PIPE_INFORMATION structures to describe each pipe in the interface.

Comments

Members that are part of this structure, but not described here, should be treated as opaque and considered to be reserved for system use.

See Also

USBD_PIPE_INFORMATION

USBD_INTERFACE_LIST_ENTRY

```
typedef struct _USBD_INTERFACE_LIST_ENTRY {
  PUSB_INTERFACE_DESCRIPTOR InterfaceDescriptor;
  PUSBD_INTERFACE_INFORMATION Interface;
} USBD_INTERFACE_LIST_ENTRY, *PUSBD_INTERFACE_LIST_ENTRY;
```

USB client drivers use this structure to create an array of interfaces to be inserted into a configuration request.

Members

InterfaceDescriptor

Points to a USB_INTERFACE_DESCRIPTOR structure that describes the interface to be added to the configuration request.

Interface

Points to a USBD_INTERFACE_INFORMATION structure that describes the properties and settings of the interface pointed to by *InterfaceDescriptor*.

Comments

This structure is used by USB clients with the routine **USBD_CreateConfiguration-RequestEx**. Clients allocate an array of these structures, one for each interface to be configured. Clients must also allocate a NULL entry in the array to be used as a terminator before calling **USBD_CreateConfigurationRequestEx**.

See Also

USBD_CreateConfigurationRequestEx

USBD_PIPE_INFORMATION

```
typedef struct _USBD_PIPE_INFORMATION {
  USHORT MaximumPacketSize ;
  UCHAR EndpointAddress ;
  UCHAR Interval ;
  USBD_PIPE_TYPE PipeType ;
  USBD_PIPE_HANDLE PipeHandle ;
  ULONG MaximumTransferSize ;
  .
  .
} USBD_PIPE_INFORMATION, *PUSBD_PIPE_INFORMATION ;
```

This structure is used by USB client drivers to hold information about a pipe from a specific interface.

Members

MaximumPacketSize

Specifies the maximum packet size, in bytes, that this pipe handles.

EndpointAddress

Specifies the bus address for this pipe.

Interval

Specifies a polling period for this pipe in milliseconds. This value is only valid if **PipeType** is set to UsbdPipeTypeInterrupt.

PipeType

Specifies what type of transfers this pipe uses. This value must be one of the following:

UsbdPipeTypeControl

Specifies that this pipe is a control pipe.

UsbdPipeTypeIsochronous

Specifies that this pipe uses isochronous transfers.

UsbdPipeTypeBulk

Specifies that this pipe uses bulk transfers.

UsbdPipeTypeInterrupt

Specifies that this pipe uses interrupt transfers. A value will be set in **Interval** to indicate how often this pipe is polled for new data.

PipeHandle

Specifies a host controller driver-defined handle that is used to access this pipe. This field should be treated as opaque.

MaximumTransferSize

Specifies the maximum size, in bytes, for a transfer request on this pipe.

Comments

Members that are part of this structure, but not described here, should be treated as opaque and considered to be reserved for system use.

USBD_ISO_PACKET_DESCRIPTOR

```
typedef struct _USBD_ISO_PACKET_DESCRIPTOR {
  ULONG Offset ;
  ULONG Length ;
  USBD_STATUS Status ;
} USBD_ISO_PACKET_DESCRIPTOR, *PUSBD_ISO_PACKET_DESCRIPTOR ;
```

This structure is used by USB client drivers to describe an isochronous transfer packet.

Members

Offset

Specifies the offset, in bytes, of the buffer for this packet from the beginning of the entire isochronous transfer buffer.

Length

Contains the number of bytes read (on return from the host controller driver) or the number of bytes to write to the isochronous pipe.

Status

Contains the status, on return from the host controller driver, of this transfer packet.

Comments

This structure is used as part of an isochronous transfer request to the host controller driver using _URB_ISOCH_TRANSFER. The **Offset** member contains the offset from the beginning of the **TransferBuffer** or **TransferBufferMDL** members of the _URB_ISOCH_ TRANSFER structure.

See Also

_URB_ISOCH_TRANSFER

PART 5

IEEE 1394 Drivers

CHAPTER 1

IEEE 1394 Bus I/O Requests

An IEEE 1394 device driver must communicate with its device by submitting IRPs down the device stack to the 1394 bus driver.

To use these I/O requests, include the header file *1394.h*, which is shipped with the Windows® 2000 DDK.

IOCTL_CLASS_1394

A IEEE 1394 driver uses the IRP_MJ_DEVICE_CONTROL IRP, with **IoControlCode** IOCTL_CLASS_1394, to communicate with the bus driver. The driver has access to all operations provided by the IEEE 1394 bus and its host controller through this request.

Input

Parameters->Others.Arguments1 points to an **IRB** structure. The **FunctionNumber** member of the IRB specifies the type of request. The **u** member of the IRB is a union that specifies the request-type-specific parameters of the request. The parameters and their meaning are documented below with each request.

Output

Parameters->Others.Arguments1 points to the **IRB** structure passed as input. As part of completing the request, the bus driver fills in certain members of the **u** member with information for the driver. The returned information is documented below with each request.

I/O Status Block

The information the bus driver returns in the I/O Status Block is documented below with each request.

REQUEST_ALLOCATE_ADDRESS_RANGE

The REQUEST_ALLOCATE_ADDRESS_RANGE request allocates addresses in the computer's IEEE 1394 address space.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            PMDL              Mdl;
            ULONG             fulFlags;
            ULONG             nLength;
            ULONG             MaxSegmentSize;
            ULONG             fulAccessType;
            ULONG             fulNotificationOptions;
            PVOID             Callback;
            PVOID             Context;
            ADDRESS_OFFSET    Required1394Offset;
            PSLIST_HEADER     FifoSListHead;
            PKSPIN_LOCK       FifoSpinLock;
            ULONG             AddressesReturned;
            PADDRESS_RANGE    p1394AddressRange;
            HANDLE            hAddressRange;
            PVOID             DeviceExtension;
        } AllocateAddressRange;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ALLOCATE_ADDRESS_RANGE

u.AllocateAddressRange.Mdl

If non-NULL, pointer to the MDL that describes the application's buffer where asynchronous operations are to be read, written, or locked. If the driver specifies **u.AllocateAddressRange.Mdl**, then **u.AllocateAddressRange.FifoSListHead** and **u.AllocateAddressRange.FifoSpinLock** must be NULL.

u.AllocateAddressRange.nLength

Specifies the number of the IEEE 1394 addresses to allocate.

u.AllocateAddressRange.MaxSegmentSize

Specifies the maximum size for each range of addresses the bus driver allocates. Use zero to indicate that the driver does not have a required maximum segment size. This member is ignored if **u.AllocateAddressRange.Required1394Offset** is non-NULL.

u.AllocateAddressRange.fulFlags

Specifies whether the array entries in p1394AddressRange use big-endian byte order. If the caller specifies BIG_ENDIAN_ADDRESS_RANGE, the array entries will be in big-endian byte order (the native byte order of the IEEE 1394 protocol), even if the local host is a little-endian machine.

u.AllocateAddressRange.fulAccessType

Specifies access type using one or more of the following flags.

Access	Description
ACCESS_FLAGS_TYPE_READ	Allocated addresses can be read.
ACCESS_FLAGS_TYPE_WRITE	Allocated addresses can be written to.
ACCESS_FLAGS_TYPE_LOCK	Allocated addresses can be the target of a lock operation.
ACCESS_FLAGS_TYPE_ BROADCAST	Allocated addresses can receive asynchronous I/O requests from any node on the bus. (By default, only the device driver's device can send requests to the allocated addresses).

u.AllocateAddressRange.fulNotificationOptions

If the device driver requests that the bus driver handle each request, and notifies the device driver upon completion, this specifies which asynchronous I/O request types will trigger the bus driver to the notify the device driver upon completion. See the Comments section for more details. The driver may specify one or more of the NOTIFY_FLAGS_AFTER_XXX flags.

Flag	Description
NOTIFY_FLAGS_NEVER	No notification.
NOTIFY_FLAGS_AFTER_READ	Notify the device driver after carrying out an asynchronous read operation.
NOTIFY_FLAGS_AFTER_WRITE	Notify the device driver after carrying out an asynchronous write operation.
NOTIFY_FLAGS_AFTER_LOCK	Notify the device driver after carrying out an asynchronous lock operation.

u.AllocateAddressRange.Callback

Pointer to a device driver callback routine. If the device driver specifies that the bus driver notify the device driver for each asynchronous I/O request, **u.AllocateAddressRange.Callback** points to the device driver's notification routine, which must have the following prototype.

```
VOID DriverNotificationRoutine(IN PNOTIFICATION_INFO );
```

If the device driver specifies that it receives no notification, and submits this request at raised IRQL through the port driver's physical mapping routine, then **u.AllocateAddressRange.Callback** points to the device driver's allocation completion routine, which must have the following prototype.

```
VOID AllocationCompletionRoutine( IN PVOID );
```

Drivers that do not request notification, and submit this request in the normal way at PASSIVE_LEVEL, must set this member to NULL.

u.AllocateAddressRange.Context

Pointer to any context data that that the device driver wants to pass for this set of addresses. If the provided callback (see previous) is a notification routine, the bus driver passes **u.AllocateAddressRange.Context** within the NOTIFICATION_INFO the parameter. If the callback is an allocation completion routine, the bus driver passes **u.AllocateAddressRange.Context** as the sole parameter to the routine.

u.AllocateAddressRange.Required1394Offset

Specifies a hard-coded address in the computer's IEEE 1394 address space. The bus driver allocates the addresses beginning at **u.AllocateAddressRange.Required1394Offset**. If no specific address is required, the driver should fill in each member of the ADDRESS_OFFSET with zero. The bus driver then chooses the addresses to allocate.

u.AllocateAddressRange.FifoSListHead

If non-NULL, specifies a properly initialized (for example, by **ExInitializeSListHead**) interlocked, singly-linked list of ADDRESS_FIFO elements. Each ADDRESS_FIFO contains an MDL. As the bus driver handles each incoming write request to the allocated addresses, it pops off the first element on the list and writes incoming data to the MDL. It then calls the driver's notification routine.

Each MDL provided must only span one page in memory. The driver can add or remove elements from the ADDRESS_FIFO list by using **ExInterlockedPushEntrySList** and **ExInterlockedPopEntrySList**.

If this member is non-NULL, the **Mdl** member of **u.AllocateAddress** range must be NULL, the **fulNotificationFlags** member must be NOTIFY_FLAGS_AFTER_WRITE (no other flags must be specified), and the driver must provide a spin lock in **FifoSpinLock**.

u.AllocateAddressRange.FifoSpinLock

If non-NULL, specifies a properly initialized spin lock (for example, by **KeInitializeSpinLock**). The spin lock will be used to serialize access to the SList provided in **u.AllocateAddressRange.FifoSListHead**.

The **u.AllocateAddressRange.FifoSpinLock** member is non-NULL if and only if **u.AllocateAddressRange.FifoSListHead** is non-NULL as well.

u.AllocateAddressRange.p1394AddressRange

Pointer to an array of ADDRESS_RANGE structures. The array must be large enough to hold the maximum number of structures the bus driver can return.

If the driver specifies a required address offset, or if the driver does not provide any backing store, the bus driver only returns one address range. If the driver provides backing store in **u.AllocateAddressRange.Mdl** the bus driver segments the allocated addresses along physical memory boundaries. If the **MaxSegmentSize** of **u.AllocateAddressRange** is 0, or if **MaxSegmentSize** is bigger than the page size, the driver can use the ALLOCATE_AND_SIZE_TO_SPAN_PAGES macro to determine the worst case. Otherwise, the maximum number of addresses ranges returned by the bus driver is **u.AllocateAddressRange.nLength/u.MaxSegmentSize**.

u.AllocateAddressRange.DeviceExtension

Reserved.

IRB Output

u.AllocateAddressRange.AddressesReturned

Specifies the number of ADDRESS_RANGE structures returned in the **p1394Address** member.

u.AllocateAddressRange.p1394AddressRange

If the request is successful, this points to an array of ADDRESS_RANGE structures that describe the set of address ranges allocated to fulfill this request.

u.AllocateAddressRange.hAddressRange

If the request is successful, this specifies the handle to use when freeing the allocated address ranges with the REQUEST_FREE_ADDRESS_RANGE request.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

Operation

If the driver provides an address in **u.AllocateAddressRange.Required1394Offset**, the bus driver allocates one contiguous range of addresses, beginning at that address. Otherwise, the bus driver allocates several ranges of addresses to fulfill the request.

Asynchronous I/O packets sent to the allocated address range are received by the bus driver. What the bus driver does with the packet depends on settings in the IRB when it is submitted.

Settings for u.AllocateAddressRange	Bus driver action upon receiving a request packet
Mdl non-NULL, and **fulNotificationFlags** is NOTIFY_FLAGS_NEVER	The bus driver transparently handles the request packet by reading or writing data using the MDL. The device driver receives no notification. For this setting, the bus driver allocates the address range asynchronously. Upon completion, it calls the allocation completion routine that the driver passed in **u.AllocateAddressRange.Callback**. As a parameter, it passes **u.AllocateAddressRange.Context**. This form of REQUEST_ALLOCATE_ADDRESS_RANGE can only be submitted in the normal way at IRQL PASSIVE_LEVEL. At raised IRQL, the driver can submit the IRB directly to the port driver through a special interface. See below for details.
Mdl non-NULL, and one or more of NOTIFY_FLAGS_AFTER_XXX are specified. (**FifoSListHead** and **FifoSpinLock** must be NULL.)	The bus driver handles the request packet by reading or writing data using the MDL. After each request type specified by NOTIFY_FLAGS_AFTER_XXX, the bus driver calls the driver's notification routine (passed in **u.AllocateAddressRange.Callback**), and passes a description of the operation in NOTIFICATION_INFO. See *NOTIFICATION_INFO* in Chapter 2 for details. Drivers can submit this form of REQUEST_ALLOCATE_ADDRESS_RANGE at any IRQL.
FifoSListHead and **FifoSpinLock** non-NULL, and the only notification flag is NOTIFY_FLAGS_AFTER_WRITE. (**Mdl** must be NULL.)	The bus driver acquires the spin lock, pops the first element off the list, and uses the MDL to handle the request packet. It then calls the driver's notification routine (passed in **u.AllocateAddressRange.Callback**), and passes a description of the operation in NOTIFICATION_INFO. See *NOTIFICATION_INFO* in Chapter 2 for details. Drivers can submit this form of REQUEST_ALLOCATE_ADDRESS_RANGE at any IRQL.

Settings for u.AllocateAddressRange	Bus driver action upon receiving a request packet
Mdl, **FifoSListHead**, and **FifoSpinLock** NULL.	The bus driver passes the request packet in the NOTIFICATION_INFO parameter of the driver's notification routine (passed in **u.AllocateAddress-Range.Callback**). The device driver must provide the response packet. See *NOTIFICATION_INFO* in Chapter 2. The NOTIFY_FLAGS_XXX flags are ignored. The bus driver ignores the setting of **u.AllocateAddress-Range.MaxSegmentSize**, and always returns a contiguous range of addresses. Drivers can submit this setting of REQUEST_ ALLOCATE_ADDRESS_RANGE at any IRQL.

REQUEST_ALLOCATE_ADDRESS_RANGE can be submitted through **IoCallDriver** at any IRQL, with one exception. If the driver receives no notification (**u.AllocateAddress-Range.Mdl** is non-NULL and **u.AllocateAddressRange.fulNotificationFlags** is NOTIFY_ FLAGS_NEVER), then the request can only be submitted through **IoCallDriver** at PASSIVE_LEVEL.

In this specific circumstance, the driver can submit the request through an alternative method, the port driver's physical mapping routine. The device driver can hand off the IRB directly to the physical mapping routine. Drivers can get a pointer to the physical mapping routine by submitting the REQUEST_GET_LOCAL_HOST_INFO bus request with **nLevel** = GET_PHYS_ADDR_ROUTINE. See *GET_LOCAL_HOST_INFO4* in Chapter 2 for details.

The following table explains how to submit the request at different IRQLs.

Notification?	DISPATCH_LEVEL	Below DISPATCH_LEVEL
yes	**IoCallDriver**	**IoCallDriver**
no	Port driver's physical mapping routine	**IoCallDriver**

REQUEST_ASYNC_LOCK

The REQUEST_ASYNC_LOCK request performs an asynchronous lock operation to the device specified.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            IO_ADDRESS    DestinationAddress;
            ULONG         nNumberOfArgBytes;
            ULONG         nNumberOfDataBytes;
            ULONG         fulTransactionType;
            ULONG         fulFlags;
            ULONG         Arguments[2];
            ULONG         DataValues[2];
            PVOID         pBuffer;
            ULONG         ulGeneration;
            UCHAR         chPriority;
            UCHAR         nSpeed;
            UCHAR         tCode;
            ULONG         Reserved;
        } AsyncLock;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ASYNC_LOCK.

u.AsyncLock.DestinationAddress

Specifies the 1394 64-bit destination address for this read operation. The driver only needs to fill in the **IA_Destination_Offset** member of **u.AsyncLock.DestinationAddress**; the bus driver will fill in the **IA_Destination_ID** member. See IO_ADDRESS for the structure description.

u.AsyncLock.nNumberOfArgBytes

Specifies the number of argument bytes used in performing this lock operation. May be zero, 4 or 8. See the **u.AsyncLock.fulTransactionType** member for details.

u.AsyncLock.nNumberOfDataBytes

Specifies the number of data bytes used in performing this lock operation. May be 4 or 8. See the **u.AsyncLock.fulTransactionType** member for details.

u.AsyncLock.fulTransactionType

Specifies which atomic transaction to execute on the 1394 node. The following function types are supported:

fulTransactionType	Description
LOCK_TRANSACTION_ MASK_SWAP	For each bit in the original value and the matching argument, reset the bit to be the same as the corresponding bit in the data value. The **nNumberOfArgBytes** and **nNumberOfDataBytes** members of **u.AsyncLock** must be the same.
LOCK_TRANSACTION_ COMPARE_SWAP	If the original value and argument match, replace the original value with the data value. The **nNumberOfArgBytes** and **nNumberOfDataBytes** members of **u.AsyncLock** must be the same.
LOCK_TRANSACTION_ FETCH_ADD	Add the data value to the original value. Big-endian addition is performed. The argument value is not used and the **nNumber-OfArgBytes** member of **u.AsyncLock** must be zero.
LOCK_TRANSACTION_ LITTLE_ADD	Add the data value to the original value. Little-endian addition is performed. The argument value is not used and the **nNumber-OfArgBytes** member of **u.AsyncLock** must be zero.
LOCK_TRANSACTION_ BOUNDED_ADD	If the original value and the argument differ, add the data value to the original value. The **nNumberOfArgBytes** and **nNumber-OfDataBytes** members of **u.AsyncLock** must be the same.
LOCK_TRANSACTION_ WRAP_ADD	If the original value and the argument differ, add the data value to original value. Otherwise, replace the original value with the data value. The **nNumberOfArgBytes** and **nNumberOfDataBytes** members of **u.AsyncLock** must be the same.

u.AsyncLock.fulFlags

Not currently used. Drivers should set this to zero.

u.AsyncLock.Arguments

Specifies the arguments used in this lock operation.

u.AsyncLock.DataValues

Specifies the data values used in this lock operation.

u.AsyncLock.pBuffer

Pointer to a buffer that receives lock data values which are returned from the node. The size of the buffer must be at least equal to the **u.AsyncLock.nNumberOfDataBytes** member.

u.AsyncLock.ulGeneration

Specifies the bus reset generation as known by the device driver who submitted this asynchronous request. If the generation count specified does not match the actual generation of the bus, then this request is returned with an error.

u.AsyncLock.chPriority

Reserved.

u.AsyncLock.nSpeed

Reserved.

u.AsyncLock.tCode

Reserved.

u.AsyncLock.Reserved

Reserved.

IRB Output

u.AsyncLock.pBuffer

Pointer to a buffer that the bus driver has filled in with the lock data values returned from the node.

I/O Status Block

If successful, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS. If **u.AsyncLock.ulGeneration** does not match the current bus reset generation count, the bus driver sets **Irp->IoStatus.Status** as STATUS_INVALID_GENERATION.

Operation

An asynchronous lock request performs an atomic operation in the node's address space. The value of the **u.AsyncLock.fulTransactionType** member determines the operation performed. The original value at the location given by the **u.AsyncLock.DestinationAddress** and the argument (in **u.AsyncLock.Arguments**) are compared, and depending on the outcome and the transaction type, the data value in **u.AsyncLock.DataValues** is used to update the original value. The new value stored at the destination address is returned in the buffer pointed to by **u.AsyncLock.pBuffer**.

REQUEST_ASYNC_READ

The REQUEST_ASYNC_READ request performs an asynchronous read operation to the device specified.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            IO_ADDRESS    DestinationAddress;
            ULONG         nNumberOfBytesToRead;
            ULONG         nBlockSize;
            ULONG         fulFlags;
            PMDL          Mdl;
            ULONG         ulGeneration;
            UCHAR         chPriority;
            UCHAR         nSpeed;
            UCHAR         tCode;
            ULONG         Reserved;
            ULONG         ElapsedTime;
        } AsyncRead;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ASYNC_READ.

u.AsyncRead.DestinationAddress

Specifies the 1394 64-bit destination address for this read operation. The driver only needs to fill in the **IA_Destination_Offset** member of **DestinationAddress**; the bus driver will fill in the **IA_Destination_ID** member. See IO_ADDRESS for the structure description.

u.AsyncRead.nNumberOfBytesToRead

Specifies the number of bytes to be read from the 1394 node.

u.AsyncRead.nBlockSize

Specifies the size of each individual block within the data stream that is read as a whole from the 1394 node. If this parameter is zero, the maximum packet size for the device and speed selected is used to issue these read requests.

u.AsyncRead.fulFlags

Specifies any non-default settings for this operation. The following flags are provided:

Flag	Description
ASYNC_FLAGS_ NONINCREMENTING	When the bus driver splits the request into blocks, begin the operation for each block at the same address, rather than treating each block as consecutive sections of the device's address space. Used only in asynchronous requests larger than **u.AsyncRead. nBlockSize** or the maximum packet size for the current speed
ASYNC_FLAGS_PING	The bus driver returns the elapsed time of the operation in **u.AsyncRead.ElapsedTime**.

u.AsyncRead.Mdl

Pointer to an MDL that describes the device driver's buffer, which receives data from the 1394 node.

u.AsyncRead.ulGeneration

Specifies the bus reset generation as known by the device driver that submits this asynchronous request. If the generation count specified does not match the actual generation of the bus, this request is returned with an error of STATUS_INVALID_GENERATION.

u.AsyncRead.chPriority

Reserved.

u.AsyncRead.nSpeed

Reserved.

u.AsyncRead.tCode

Reserved.

u.AsyncRead.Reserved

Reserved.

IRB Output

u.AsyncRead.ElapsedTime

If the driver specifies the ASYNC_FLAGS_PING flag, the bus driver returns the time required, in nanoseconds, to complete the read request.

I/O Status Block

If successful, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS. If **u.AsyncLock.ulGeneration** does not match the current bus reset generation count, the bus driver sets **Irp->IoStatus.Status** as STATUS_INVALID_GENERATION.

Operation

A REQUEST_ASYNC_READ request will read from the device's address space, beginning at the **AddressOffset** member of **u.AsyncRead.DestinationAddress**. The buffer will be broken up into blocks, and one block will be read per transaction. If the ASYNC_FLAGS_NONINCREMENTING flag is set, the driver reads each block beginning at **u.AsyncRead.DestinationAddress**; otherwise it will read each block from successive regions in the device's memory address space.

The *IEEE 1394-1995 Specification* constrains the size of reads to be ASYNC_PAYLOAD_xxx_RATE, where xxx is the approximate connection speed in megabits per second. (The speeds allowed at the time of this writing are 100, 200, and 400 Mb/s.) If the block size exceeds the maximum payload size, the payload size will be used as the block size.

The size of packets may also be constrained by the device itself. The device reports the maximum packet size in the MAX_REC field of its configuration ROM. If this value is smaller than requested block size and the maximum payload size, the bus driver uses this as the block size.

REQUEST_ASYNC_STREAM

The REQUEST_ASYNC_STREAM request writes packets to an isochronous channel, during the asynchronous phase of the IEEE 1394 bus.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
```

```
    union {
        struct {
            ULONG           nNumberOfBytesToStream;
            ULONG           fulFlags;
            PMDL            Mdl;
            ULONG           ulTag;
            ULONG           nChannel;
            ULONG           ulSynch;
            ULONG           Reserved;
            UCHAR           nSpeed;
        } AsyncStream;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ASYNC_STREAM.

u.AsyncStream.nNumberOfBytesToStream

Specifies the number of bytes to write.

u.AsyncStream.fulFlags

Reserved. Drivers must set this to zero.

u.AsyncStream.Mdl

Specifies the source buffer.

u.AsyncStream.ulTag

Specifies the Tag field for any packets generated from this request.

u.AsyncStream.nChannel

Specifies the channel to which the data will be written.

u.AsyncStream.ulSynch

Specifies the Sy field for any packets generated from this request.

u.AsyncStream.Reserved

Reserved.

u.AsyncStream.nSpeed

Specifies the transfer rate. The possible speed values are SPEED_FLAGS_xxx, where xxx is the (approximate) transfer rate in megabits per second. At the time of this writing, existing hardware currently supports transfer rates of 100, 200, and 400 Mb/sec.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

Operation

Since the packet is sent during the asynchronous phase of the bus cycle, it does not have a guaranteed bandwidth and, therefore, it lacks a guaranteed delivery time.

REQUEST_ASYNC_WRITE

The REQUEST_ASYNC_WRITE request performs an asynchronous write operation to the device specified.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            IO_ADDRESS    DestinationAddress;
            ULONG         nNumberOfBytesToWrite;
            ULONG         nBlockSize;
            ULONG         fulFlags;
            PMDL          Mdl;
            ULONG         ulGeneration;
            UCHAR         chPriority;
            UCHAR         nSpeed;
            UCHAR         tCode;
            ULONG         Reserved;
        } AsyncWrite;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ASYNC_WRITE

u.AsyncWrite.DestinationAddress

Specifies the 1394 64-bit destination address for this read operation. The driver only must fill in the **IA_Destination_Offset** member of **u.AsyncWrite.DestinationAddress**; the bus driver will fill in the **IA_Destination_ID** member. See IO_ADDRESS for the structure description.

u.AsyncWrite.nNumberOfBytesToWrite

Specifies the number of bytes to write to the 1394 node.

u.AsyncWrite.nBlockSize

Specifies the size of each individual block within the data stream that is written as a whole to the node. If this parameter is zero, then the maximum packet size for the speed selected is used in breaking up these write requests.

u.AsyncWrite.fulFlags

Specifies any non-default settings for this operation. The following flags are provided:

Flag	Description
ASYNC_FLAGS_ NONINCREMENTING	Do not increment 1394 address during asynchronous operation. This flag is set only in large asynchronous requests (that is, those greater than asynchronous packet size).
ASYNC_FLAGS_NO_STATUS	Always return success from the write operation, whether the write succeeds or fails.

Use the bit-wise operator OR to combine the settings.

u.AsyncWrite.Mdl

Pointer to an MDL that describes the device driver's buffer, which receives data from the 1394 node.

u.AsyncWrite.ulGeneration

Specifies the bus reset generation as known by the device driver that submitted this asynchronous request. If the generation count specified does not match the actual generation of the bus, this request is returned with an error.

u.AsyncWrite.chPriority

Reserved.

u.AsyncWrite.nSpeed

Reserved.

u.AsyncWrite.tCode

Reserved.

u.AsyncWrite.Reserved

Reserved.

I/O Status Block

If successful, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS. If **u.AsyncLock.ulGeneration** does not match the current bus reset generation count, the bus driver sets **Irp->IoStatus.Status** to STATUS_INVALID_GENERATION.

Operation

The REQUEST_ASYNC_WRITE request writes the data buffer into the device's address space, beginning at the **u.AsyncWrite.AddressOffset** member of **u.AsyncWrite.DestinationAddress**. The buffer is broken up into blocks, and one block is written for each transaction. If the ASYNC_FLAGS_NONINCREMENTING flag is set, the bus driver will write each block beginning at **u.AsyncWrite.DestinationAddress**, otherwise it will write the buffer to consecutive addresses in the memory space of the device.

The *IEEE 1394-1995 Specification* constrains the size of write operations to be ASYNC_PAYLOAD_xxx_RATE, where xxx is the approximate connection speed in megabits per second. (The speeds allowed at the time of this writing are 100, 200, and 400 Mb/s.). If the block size exceeds the maximum payload size, the payload size will be used as the block size.

The size of packets may also be constrained by the device itself. The device reports the maximum packet size in the MAX_REC field of its configuration ROM. If this value is smaller than requested block size and the maximum payload size, the bus driver uses this as the block size.

The ASYNC_FLAGS_NO_STATUS flag should only be set if the driver has a higher-level error recovery protocol, and should only be used for quadlet-sized writes.

REQUEST_BUS_RESET

The REQUEST_BUS_RESET request initiates an IEEE 1394 bus reset.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            ULONG    fulFlags;
        } BusReset;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_BUS_RESET.

u.BusReset.fulFlags

Specifies flags for the bus reset. Set the flag BUS_RESET_FLAGS_FORCE_ROOT to make the local node the root node.

REQUEST_BUS_RESET_NOTIFICATION

The REQUEST_BUS_RESET_NOTIFICATION request registers (or de-registers) a notification routine to be executed for IEEE 1394 bus resets.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            ULONG                          fulFlags;
            PBUS_BUS_RESET_NOTIFICATION ResetRoutine;
            PVOID                          ResetContext;
        } BusResetNotification;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_BUS_RESET_NOTIFICATION.

u.BusResetNotification.fulFlags

Specifies whether a callback should be registered or deactivated. Use REGISTER_NOTIFICATION_ROUTINE to register **ResetRoutine** as the callback. Use DEREGISTER_NOTIFICATION_ROUTINE to deactivate any previously registered callback.

u.BusResetNotification.ResetRoutine

Pointer to the notification routine for bus resets. The notification routine parameters follow this prototype:

```
void BusResetNotificationRoutine(IN PVOID Context);
```

u.BusResetNotification.ResetContext

Specifies the argument to be passed to the notification routine.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

Operation

The notification routine is called at DISPATCH_LEVEL.

REQUEST_CONTROL

The REQUEST_CONTROL request submits a vendor-specific control request to the port driver.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
```

```
        struct {
            ULONG    ulIoControlCode;
            PMDL     pInBuffer;
            ULONG    ulInBufferLength;
            PMDL     pOutBuffer;
            ULONG    ulOutBufferLength;
            ULONG    BytesReturned;
        } Control;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_CONTROL.

u.Control.ulIoControlCode

Specifies the control code used in this request. Vendors should make these control codes unique, so that they do not overlap.

u.Control.pInBuffer

Pointer to an MDL that describes the input buffer. The input buffer contains user-defined information.

u.Control.ulInBufferLength

Specifies the length of the input buffer.

u.Control.pOutBuffer

Pointer to an MDL that describes the output buffer. The output buffer contains user-defined information.

u.Control.ulOutBufferLength

Specifies the length of the output buffer.

IRB Output

u.Control.pOutBuffer

The bus driver puts the output of the control request in the buffer described by this MDL.

u.Control.BytesReturned

The length of any data that is returned in **pOutBuffer**.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

Operation

Vendors interested in defining their own control codes for their host controller should contact Microsoft for a unique **IoControlCode**. This ensures that the **IoControlCode** will not overlap with the control codes used by other hardware and software vendors.

Drivers that submit this request must be running at an IRQL of PASSIVE_LEVEL.

REQUEST_FREE_ADDRESS_RANGE

The REQUEST_FREE_ADDRESS_RANGE request releases a 1394 address allocated by REQUEST_ALLOCATE_ADDRESS_RANGE.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            ULONG              nAddressesToFree;
            PADDRESS_RANGE     p1394AddressRange;
            PHANDLE            pAddressRange;
            PVOID              Reserved;
        } FreeAddressRange;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_FREE_ADDRESS_RANGE.

u.FreeAddressRange.nAddressesToFree

Specifies the number of entries in **pAddressRange.**

u.FreeAddressRange.p1394AddressRange

Specifies a pointer to an array of ADDRESS_RANGE data structures to be released. These address ranges were returned in a prior successful call to **AllocateAddressRange**.

u.FreeAddressRange.pAddressRange

Pointer to the array of handles to free. These handles were returned in a prior successful call to **AllocateAddressRange**.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

REQUEST_GET_ADDR_FROM_DEVICE_OBJECT

The REQUEST_GET_ADDR_FROM_DEVICE_OBJECT request returns a 1394 node address.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            ULONG           fulFlags;
            NODE_ADDRESS    NodeAddress;
        } Get1394AddressFromDeviceObject;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_GET_ADDR_FROM_DEVICE_OBJECT.

u.Get1394AddressFromDeviceObject.fulFlags

Specifies which device's node address we are querying. Zero indicates the calling device. USE_LOCAL_NODE indicates the local host controller.

IRB Output

u.Get1394AddressFromDeviceObject.NodeAddress

Contains the NODE_ADDRESS structure describing the device's node address.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

REQUEST_GET_CONFIGURATION_INFO

The REQUEST_GET_CONFIGURATION_INFO request retrieves configuration information for a device. This information is gathered from the device's IEEE 1394 standard configuration ROM.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            PCONFIG_ROM      ConfigRom;
            ULONG            UnitDirectoryBufferSize;
            PVOID            UnitDirectory;
            IO_ADDRESS       UnitDirectoryLocation;
            ULONG            UnitDependentDirectoryBufferSize;
            PVOID            UnitDependentDirectory;
            IO_ADDRESS       UnitDependentDirectoryLocation;
            ULONG            VendorLeafBufferSize;
            PTEXTUAL_LEAF    VendorLeaf;
            ULONG            ModelLeafBufferSize;
            PTEXTUAL_LEAF    ModelLeaf;
        } GetConfigurationInformation;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_GET_CONFIGURATION_INFO.

u.GetConfigurationInformation.ConfigRom

Pointer to the buffer that the bus driver will use to store a copy of the device's configuration ROM. The configuration ROM is defined by the *IEEE 1394 Specification.*

u.GetConfigurationInformation.UnitDirectoryBufferSize

Specifies the size of the buffer pointed to by the **UnitDirectory** member of **u.GetConfigurationInformation**. If the **UnitDirectoryBufferSize**, **UnitDependentDirectoryBufferSize**, **VendorLeafBufferSize**, and **ModelLeafBufferSize** members of **u.Get ConfigurationInformation** are all zero, on completion the bus driver fills in this member with the minimum buffer size needed to hold all the information.

u.GetConfigurationInformation.UnitDirectory

Pointer to where the bus driver will return the unit directory. See the *IEEE 1394-1995 Specification* for a description of the internals of the unit directory.

u.GetConfigurationInformation.UnitDependentDirectoryBufferSize

Specifies the size of the buffer pointed to by **UnitDependentDirectory** member of **u.GetConfigurationInformation**. If the **UnitDirectoryBufferSize**, **UnitDependentDirectoryBufferSize**, **VendorLeafBufferSize**, and **ModelLeafBufferSize** members of **u.GetConfigurationInformation** are all zero, on completion the bus driver fills in this member with the minimum buffer size needed to hold all the information.

u.GetConfigurationInformation.UnitDependentDirectory

Pointer to a buffer that will receives the unit dependent directory, as defined by the *IEEE 1394-1995 Specification.* See the specification for a description of the internals of the unit directory.

u.GetConfigurationInformation.VendorLeafBufferSize

Specifies the size of the buffer pointed to by **VendorLeaf** member of **u.GetConfigurationInformation**. If the **UnitDirectoryBufferSize**, **UnitDependentDirectoryBufferSize**, **VendorLeafBufferSize**, and **ModelLeafBufferSize** members of **u.GetConfigurationInformation** are all zero, on completion the bus driver fills in this member with the minimum buffer size needed to hold all the information.

u.GetConfigurationInformation.VendorLeaf

Pointer to a buffer to receive the vendor leaf TEXTUAL_LEAF structure, which describes the device vendor.

u.GetConfigurationInformation.ModelLeafBufferSize

Specifies the size of the buffer pointed to by **ModelLeaf** member of **u.GetConfiguration-Information**. If the **UnitDirectoryBufferSize**, **UnitDependentDirectoryBufferSize**, **VendorLeafBufferSize**, and **ModelLeafBufferSize** members of **u.GetConfiguration-Information** are all zero, on completion the bus driver fills in this member with the minimum buffer size needed to hold all the information.

u.GetConfigurationInformation.ModelLeaf

Pointer to a buffer to receive the model leaf TEXTUAL_LEAF structure, which describes the device model type.

IRB Output

u.GetConfigurationInformation.ConfigRom

If successful, the bus driver returns the configuration ROM in this buffer. If the **Unit-DirectoryBufferSize**, **UnitDependentDirectoryBufferSize**, **VendorLeafBufferSize**, and **ModelLeafBufferSize** members of **u.GetConfigurationInformation** are all zero, the bus driver does not use this member.

u.GetConfigurationInformation.UnitDirectoryBufferSize

If the device driver passed zero for the **UnitDirectoryBufferSize**, **UnitDependent-DirectoryBufferSize**, **VendorLeafBufferSize**, and **ModelLeafBufferSize** members of **u.GetConfigurationInformation**, the bus driver returns the size of the unit directory in this member.

u.GetConfigurationInformation.UnitDirectory

The bus driver returns the unit directory in this buffer, up the number of bytes specified by the device driver in **u.GetConfigurationInformation.UnitDirectoryBufferSize**.

u.GetConfigurationInformation.UnitDirectoryLocation

Specifies the address in the device's memory space where the unit directory begins. This parameter is useful when pointers are embedded within the unit directory, as these pointers are expressed in quadlet offsets from the current position. Only the **IA_Destination_Offset** member of the IO_ADDRESS data structure is valid.

u.GetConfigurationInformation.UnitDependentDirectoryBufferSize

If the device driver passed zero for the **UnitDirectoryBufferSize**, **UnitDependent-DirectoryBufferSize**, **VendorLeafBufferSize**, and **ModelLeafBufferSize** members of **u.GetConfigurationInformation**, the bus driver returns the size of the unit dependent directory in this member.

u.GetConfigurationInformation.UnitDependentDirectory

The bus driver returns the unit dependent directory in this buffer, up the number of bytes specified by the device driver in **u.GetConfigurationInformation.UnitDependent-DirectoryBufferSize**.

u.GetConfigurationInformation.UnitDependentDirectoryLocation

Specifies the 48-bit address where the unit-dependent directory begins. This parameter is useful when pointers are embedded in the unit-dependent directory, since these pointers are expressed in quadlet offsets from the current position. See IO_ADDRESS for the structure description.

u.GetConfigurationInformation.VendorLeafBufferSize

If the device driver passed zero for the **UnitDirectoryBufferSize**, **UnitDependent-DirectoryBufferSize**, **VendorLeafBufferSize**, and **ModelLeafBufferSize** members of **u.GetConfigurationInformation**, the bus driver returns the size of the vendor leaf in this member.

u.GetConfigurationInformation.VendorLeaf

The bus driver returns the unit directory in this buffer, up the number of bytes specified by the device driver in **u.GetConfigurationInformation.VendorLeafBufferSize**.

u.GetConfigurationInformation.ModelLeafBufferSize

If the device driver passed zero for the **UnitDirectoryBufferSize**, **UnitDependent-DirectoryBufferSize**, **VendorLeafBufferSize**, and **ModelLeafBufferSize** members of **u.GetConfigurationInformation**, the bus driver returns the size of the model leaf in this member.

u.GetConfigurationInformation.ModelLeaf

The bus driver returns the unit directory in this buffer, up the number of bytes specified by the device driver in **u.GetConfigurationInformation.ModelLeafBufferSize**.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

REQUEST_GET_GENERATION_COUNT

The REQUEST_GET_GENERATION_COUNT request retrieves the current bus reset generation. This count is incremented every time the bus is reset. This generation count must be passed in each asynchronous read, write, and lock request that a device driver submits to the 1394 bus driver.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            ULONG    GenerationCount;
        } GetGenerationCount;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_GET_GENERATION_COUNT.

IRB Output

u.GetGenerationCount.GenerationCount

Specifies the current generation count.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

REQUEST_GET_LOCAL_HOST_INFO

The REQUEST_GET_LOCAL_HOST_INFO request returns information about the local node's host controller.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            ULONG    nLevel;
```

```
            PVOID    Information;
        } GetLocalHostInformation;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_GET_LOCAL_HOST_INFO.

u.GetLocalHostInformation.nLevel

Specifies what level of information is desired from this call. The following flags are provided.

Flag	Description
GET_HOST_UNIQUE_ID	Requests the port driver to return the 64-bit unique identifier.
GET_HOST_CAPABILITIES	Requests the port driver to return the host controller's capability flags.
GET_POWER_SUPPLIED	Requests the port driver to return the bus' power characteristics.
GET_PHYS_ADDR_ROUTINE	Requests the port driver to return the host controller's physical address mapping function.
GET_HOST_CONFIG_ROM	Requests the port driver to return the host controller's configuration ROM.
GET_HOST_CSR_CONTENTS	Requests the port driver to return the speed or topology maps from the host controller's CSR. See the IEEE 1394 Specification for a description of CSRs.

u.GetLocalHostInformation.Information

Pointer to an information block to be filled in, depending on what level of information is desired. Each block has its own particular structure:

Flag	Structure
GET_HOST_UNIQUE_ID	GET_LOCAL_HOST_INFO1
GET_HOST_CAPABILITIES	GET_LOCAL_HOST_INFO2
GET_POWER_SUPPLIED	GET_LOCAL_HOST_INFO3
GET_PHYS_ADDR_ROUTINE	GET_LOCAL_HOST_INFO4
GET_HOST_CONFIG_ROM	GET_LOCAL_HOST_INFO5
GET_HOST_CSR_CONTENTS	GET_LOCAL_HOST_INFO6

IRB Output

u.GetLocalHostInformation.Information

Pointer to the information block, with the requested information filled in.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

REQUEST_GET_SPEED_BETWEEN_DEVICES

The REQUEST_GET_SPEED_BETWEEN_DEVICES request returns the maximum (simultaneous) transfer speed that can be used from one source device to a set of destination devices.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            ULONG             fulFlags;
            ULONG             ulNumberOfDestinations;
            PDEVICE_OBJECT    hDestinationDeviceObjects[64];
            ULONG             fulSpeed;
        } GetMaxSpeedBetweenDevices;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_GET_SPEED_BETWEEN_DEVICES

u.GetMaxSpeedBetweenDevices.fulFlags

Specifies the source device. Zero indicates the calling device. USE_LOCAL_NODE indicates the computer itself.

u.GetMaxSpeedBetweenDevices.ulNumberOfDestinations

Specifies the number of destination devices.

u.GetMaxSpeedBetweenDevices.hDestinationDeviceObjects

Pointer to an array of the device objects of the destination devices.

IRB Output

u.GetMaxSpeedBetweenDevices.fulSpeed

Specifies the maximum possible transaction speed between the source device and the set of destination devices. The value returned is the maximum speed supported by all the devices simultaneously. The possible speed values are SPEED_FLAGS_xxx, where xxx is the (approximate) transfer rate in megabits per second. At the time of this writing, existing hardware supports transfer rates of 100, 200, and 400 Mb/sec.

I/O Status Block

If successful, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS.

REQUEST_GET_SPEED_TOPOLOGY_MAPS

The REQUEST_GET_SPEED_TOPOLOGY_MAPS request returns the IEEE 1394 bus speed and topology maps. On Windows 2000, this request is obsolete.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            PSPEED_MAP        SpeedMap;
            PTOPOLOGY_MAP     TopologyMap;
        } GetSpeedTopologyMaps;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_GET_SPEED_TOPOLOGY_MAPS.

IRB Output

u.GetSpeedTopologyMaps.SpeedMap

Pointer to the bus' SPEED_MAP structure. This member is filled on completion.

u.GetSpeedTopologyMaps.TopologyMap

Pointer to the bus' TOPOLOGY_MAP structure. The topology map will be in big-endian, irrespective of the byte order of the local node. This member is filled on completion.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

Operation

On Windows 2000, use GET_LOCAL_HOST_INFO with **u.GetLocalHostInformation.nLevel** = GET_HOST_CSR_CONTENTS.

REQUEST_ISOCH_ALLOCATE_BANDWIDTH

The REQUEST_ISOCH_ALLOCATE_BANDWIDTH request allocates isochronous bandwidth to be used in subsequent operations.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
   ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            ULONG     nMaxBytesPerFrameRequested;
            ULONG     fulSpeed;
            HANDLE    hBandwidth;
            ULONG     BytesPerFrameAvailable;
            ULONG     SpeedSelected;
        } IsochAllocateBandwidth;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_ALLOCATE_BANDWIDTH

u.IsochAllocateBandwidth.nMaxBytesPerFrameRequested

Specifies the bandwidth requested, in bytes per isochronous frame. If the host controller is configured to strip away the packet headers, the device driver does not need to include the packet header size in the number of bytes requested. The driver also does not need to round the value up to the nearest quadlet.

u.IsochAllocateBandwidth.fulSpeed

Specifies the connection speed to use in allocating bandwidth. The possible speed values are SPEED_FLAGS_xxx, where xxx is the (approximate) transfer rate in megabits per second. Currently, existing hardware supports transfer rates of 100, 200, and 400 Mb/sec.

IRB Output

u.IsochAllocateBandwidth.hBandwidth

Specifies the handle to use to refer to the bandwidth resource.

u.IsochAllocateBandwidth.BytesPerFrameAvailable

Specifies the bytes per frame that are available after the allocation attempt. Drivers should not rely on this bandwidth being available, since another device may allocate or deallocate bandwidth at any time. The bus driver fills in this member, even if the request fails.

u.IsochAllocateBandwidth.SpeedSelected

Specifies the actual speed selected in allocating bandwidth. The value is one of SPEED_FLAGS_xxx (see the **fulSpeed** member description above).

I/O Status Block

If the bus driver successfully allocates the bandwidth, it sets Irp->IoStatus.Status to STATUS_SUCCESS. If it defers the request, it returns STATUS_PENDING. Otherwise it sets the appropriate error status.

Operation

Once the device driver no longer needs allocated bandwidth, it must deallocate it with the REQUEST_ISOCH_FREE_BANDWIDTH request.

On Windows 98 RTM, the caller must be running at IRQL PASSIVE_LEVEL. On Windows 98 SP1, Windows 2000, or later, this request may be submitted at all IRQL levels.

REQUEST_ISOCH_ALLOCATE_CHANNEL

The REQUEST_ISOCH_ALLOCATE_CHANNEL request allocates an isochronous channel to be used in subsequent operations.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            ULONG            nRequestedChannel;
            ULONG            Channel;
            LARGE_INTEGER    ChannelsAvailable;
        } IsochAllocateChannel;
          .
          .
          .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_ALLOCATE_CHANNEL

u.IsochAllocateChannel.nRequestedChannel

Specifies the particular channel to allocate, or ISOCH_ANY_CHANNEL for an arbitrary channel. Most drivers should use ISOCH_ANY_CHANNEL.

IRB Output

u.IsochAllocateChannel.Channel

Specifies the channel allocated, if the request succeeds.

u.IsochAllocateChannel.ChannelsAvailable

Specifies a bitmap of the available isochronous channels after the channel allocation attempt. Drivers should not rely on this information, since another device may allocate or deallocate channels at any time. The bus driver fills in this member, even if the request fails.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS if it successfully allocates the channel, or otherwise to the appropriate error status.

Operation

Once a channel is no longer needed, it should be deallocated using the REQUEST_ISOCH_FREE_CHANNEL request.

Drivers that submit this request must be running at an IRQL of PASSIVE_LEVEL.

REQUEST_ISOCH_ALLOCATE_RESOURCES

The REQUEST_ISOCH_ALLOCATE_RESOURCES request allocates a resource handle for transactions over a given isochronous channel. The device driver will use the resource handle to attach and detach data buffers for isochronous transactions on that channel.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            ULONG     fulSpeed;
            ULONG     fulFlags;
            ULONG     nChannel;
            ULONG     nMaxBytesPerFrame;
            ULONG     nNumberOfBuffers;
            ULONG     nMaxBufferSize;
            ULONG     nQuadletsToStrip;
            HANDLE    hResource;
        } IsochAllocateResources;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_ALLOCATE_RESOURCES

u.IsochAllocateResources.fulSpeed

Specifies the connection speed to use for communication on the channel. The possible speed values are SPEED_FLAGS_xxx, where xxx is the (approximate) transfer rate in megabits per second. Currently, existing hardware supports transfer rates of 100, 200, and 400 Mb/sec.

u.IsochAllocateResources.fulFlags

Specifies how the bus driver should use any buffers attached to the resource handle. Many of the flags specify how the bus driver should configure the IEEE host controller for DMA from or to attached buffers.

Flag	Description
RESOURCE_USED_IN_LISTENING	Attached buffers will be used to read data from an isochronous channel. Set this if the resource handle will be used in a REQUEST_ISOCH_LISTEN request.
RESOURCE_USED_IN_TALKING	Attached buffers will be used to write data to an isochronous channel. Set this if the resource handle will be used in a REQUEST_ISOCH_TALK request.
RESOURCE_BUFFERS_CIRCULAR	When the host controller uses up the current supply of attached buffers, it will continue the operation beginning again with the first buffer attached. When the isochronous transaction is in progress, the device driver must not attach or detach buffers on the resource handle.
RESOURCE_STRIP_ADDITIONAL_QUADLETS	The bus driver will configure the host controller to strip additional quadlets from incoming isochronous packets. The number of quadlets to be stripped is specified in **nQuadletsToStrip**.
RESOURCE_SYNCH_ON_TIME	The bus driver will configure the host controller to synchronize the beginning of the isochronous transaction to the CYCLE_TIME specified in the StartTime member of the request's IRB. See REQUEST_ISOCH_LISTEN or REQUEST_ISOCH_TALK.
RESOURCE_USE_PACKET_BASED	Used to switch to packet-based transfer, rather than the default. The default is stream-based transfer, unless the host controller only supports packet-based DMA.

u.IsochAllocateResources.nChannel

Specifies the isochronous channel for all transactions involving the resource handle allocated by this request.

u.IsochAllocateResources.nMaxBytesPerFrame

Specifies the expected maximum isochronous frame size while transmitting and receiving on the channel.

u.IsochAllocateResources.nNumberOfBuffers

Specifies one more than the maximum expected number of buffers that are attached to the resource handle at any given time. If the RESOURCE_BUFFERS_CIRCULAR flag is specified, then **nNumberOfBuffers** specifies the total number of buffers attached to the resource handle, instead of the average number at any given time.

u.IsochAllocateResources.nMaxBufferSize

Specifies the maximum size of the buffers that are attached to the resource handle.

u.IsochAllocateResources.nQuadletsToStrip

Specifies the number of quadlets to strip from the beginning of every packet in an incoming isochronous stream. This parameter is ignored unless the device driver sets the RESOURCE_STRIP_ADDITIONAL_QUADLETS flag in **fulFlags**.

IRB Output

u.IsochAllocateResources.hResource

Specifies the resource handle the device driver uses to attach or detach data buffers for isochronous transactions on the channel specified in **nChannel**.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS, if it processes the request successfully, or to the appropriate error code.

Operation

Not all RESOURCE_XXX flags are supported. Some require hardware support from the IEEE 1394 host controller. The device driver can use the REQUEST_GET_LOCAL_HOST_INFO request, with **nLevel** = GET_HOST_CAPABILITIES, to determine which RESOURCE_XXX flags are supported. The bus driver returns a pointer to a GET_LOCAL_HOST_INFO2 structure, whose **HostCapabilities** member contains flags

that determine which flags the host controller supports. The following table lists which RESOURCE_XXX flags require hardware support, and the corresponding **HostCapabilities** flag the driver should check:

RESOURCE_XXX flags	HostCapabilities flag
RESOURCE_STRIP_ADDITIONAL_QUADLETS	HOST_INFO_SUPPORTS_ISOCH_STRIPPING
RESOURCE_SYNCH_ON_TIME	HOST_INFO_SUPPORTS_START_ON_CYCLE
RESOURCE_USE_PACKET_BASED	HOST_INFO_PACKET_BASED

The default method of transmission for isochronous reads is stream-based: data is read until it fills the buffer, and then the bus driver begins filling the next buffer. If the RESOURCE_USE_PACKET_BASED flag is set, the bus driver uses a packet-based method of storing data: each packet is put in its own buffer and no attempt is made to fill each buffer. A particular host controller may only support packet-based or stream-based reads. Drivers should use the REQUEST_GET_LOCAL_HOST_INFO request to determine what the host controller supports.

Not all host controllers automatically strip off the packet header. Use the REQUEST_GET_LOCAL_HOST_INFO request to determine if the host controller automatically strips the packet header. This request will also determine if the host controller can be configured to strip quadlets from the beginning of each packet. Many host controllers can be configured to automatically strip off the packet header. Try setting the RESOURCE_STRIP_ADDITIONAL_QUADLETS flag and setting **nQuadletsToStrip** to 1.

Set the RESOURCE_SYNCH_ON_TIME to synchronize the beginning of I/O to the **StartTime** member of the REQUEST_ISOCH_LISTEN or REQUEST_ISOCH_TALK request. Use the REQUEST_GET_LOCAL_HOST_INFO request to determine if the host controller supports synchronization on an isochronous cycle time. Additional synchronization options can be set with each buffer attached; see the ISOCH_DESCRIPTOR member of the REQUEST_ISOCH_ATTACH_BUFFERS for details.

Drivers that submit this request must be running at an IRQL of PASSIVE_LEVEL.

REQUEST_ISOCH_ATTACH_BUFFERS

The REQUEST_ISOCH_ATTACH_BUFFERS request attaches an array of buffers to an isochronous resource handle.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            HANDLE              hResource;
            ULONG               nNumberOfDescriptors;
            PISOCH_DESCRIPTOR   pIsochDescriptor;
        } IsochAttachBuffers;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_ATTACH_BUFFERS

u.IsochAttachBuffers.hResource

Specifies the resource handle to attach buffers to.

u.IsochAllocateResources.nNumberOfDescriptors

Specifies the number of elements in the **pIsochDescriptor** array.

u.IsochAllocateResources.pIsochDescriptor

Pointer to an array of ISOCH_DESCRIPTOR structures that describe the buffers to be attached, and the parameters that specify how each buffer is to be used.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS, if it processes the request successfully, or to the appropriate error code.

Operation

Once a buffer is no longer in use, the device driver can detach it from the resource handle by using the REQUEST_ISOCH_DETACH_BUFFERS request.

REQUEST_ISOCH_DETACH_BUFFERS

The REQUEST_ISOCH_DETACH_BUFFERS request detaches an array of buffers from an isochronous resource handle.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            HANDLE              hResource;
            ULONG               nNumberOfDescriptors;
            PISOCH_DESCRIPTOR   pIsochDescriptor;
        } IsochDetachBuffers;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_DETACH_BUFFERS

u.IsochAttachBuffers.hResource

Specifies the resource handle to detach buffers from.

u.IsochAllocateResources.nNumberOfDescriptors

Specifies the number of elements in the **pIsochDescriptor** array.

u.IsochAllocateResources.pIsochDescriptor

Pointer to an array of ISOCH_DESCRIPTOR structures that describe the buffers to be detached. The device driver should use the same ISOCH_DESCRIPTOR structure for a buffer that it used to attach the buffer.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS, if it processes the request successfully, or to the appropriate error code.

REQUEST_ISOCH_FREE_BANDWIDTH

The REQUEST_ISOCH_FREE_BANDWIDTH request releases isochronous bandwidth that was allocated through a REQUEST_ISOCH_ALLOCATE_BANDWIDTH request.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
   ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            HANDLE    hBandwidth;
        } IsochFreeBandwidth;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_FREE_BANDWIDTH

u.IsochFreeBandwidth.hBandwidth

Specifies the bandwidth handle to release.

I/O Status Block

If successful, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS, and the isochronous bandwidth is returned to the pool of available bandwidth.

Operation

Drivers that submit this request must be running at an IRQL of PASSIVE_LEVEL.

REQUEST_ISOCH_FREE_CHANNEL

The REQUEST_ISOCH_FREE_CHANNEL request releases an isochronous channel that was allocated through a REQUEST_ISOCH_ALLOCATE_CHANNEL request.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
   ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            ULONG    nChannel;
        } IsochFreeChannel;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_FREE_CHANNEL

u.IsochFreeChannel.nChannel

Specifies which allocated channel to release.

I/O Status Block

If successful, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS, and the isochronous channel is returned to the pool of available channels.

Operation

Drivers that submit this request must be running at an IRQL of PASSIVE_LEVEL.

REQUEST_ISOCH_FREE_RESOURCES

The REQUEST_ISOCH_FREE_RESOURCES request releases the resource handle that was allocated through the REQUEST_ISOCH_ALLOCATE_RESOURCES request.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            HANDLE    hResource;
        } IsochFreeResources;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_FREE_RESOURCES

u.IsochFreeResources.hResource

Specifies the resource handle to release.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success. All isochronous buffers that were attached to this resource must be detached prior to issuing this call. If a device driver attempts to free a resource handle with isochronous buffers still attached to it, the handle is not freed and the bus driver returns STATUS_INVALID_PARAMETER instead.

Operation

Drivers that submit this request must be running at an IRQL of PASSIVE_LEVEL.

REQUEST_ISOCH_LISTEN

The device driver issues a REQUEST_ISOCH_LISTEN request to read from an isochronous channel.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            HANDLE        hResource;
            ULONG         fulFlags;
            CYCLE_TIME    StartTime
        } IsochListen;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_LISTEN

u.IsochListen.hResource

Specifies the resource handle to use in reading data.

u.IsochListen.fulFlags

Reserved. Device drivers must set this to zero.

u.IsochListen.StartTime

Specifies the cycle time to begin reading data. This member is used only if the driver specified the RESOURCE_SYNCH_ON_TIME flag when it allocated the resource handle passed in **u.IsochListen.hResource**. (The timing resolution is per isochronous cycle, so the **CycleOffset** member of **StartTime** is not used.)

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS when it has successfully processed the request, or to the appropriate error code.

Operation

The bus driver completes this request once it has successfully scheduled the isochronous listen operation. It does not wait for the data transfer to finish, or even begin, before it completes the request.

Listening on a channel may begin immediately, or it may be synchronized to an event. If the driver set the RESOURCE_SYNCH_ON_TIME flag on the REQUEST_ISOCH_ALLOCATE_RESOURCES request that returned the resource handle, then the listen will begin on the cycle count specified in **StartTime**. Additional synchronization options may be set for each buffer within that buffer's ISOCH_DESCRIPTOR structure.

REQUEST_ISOCH_QUERY_CYCLE_TIME

The ISOCH_QUERY_CURRENT_CYCLE_TIME request returns the current isochronous cycle time.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
   ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            CYCLE_TIME    CycleTime;
        } IsochQueryCurrentCycleTime;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_QUERY_CYCLE_TIME

IRB Output

u.IsochQueryCurrentCycleTime.CycleTime

On success, specifies the current isochronous cycle time. See the CYCLE_TIME entry for details.

I/O Status Block

If successful, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS.

REQUEST_ISOCH_QUERY_RESOURCES

The ISOCH_QUERY_RESOURCES request returns the bandwidth and channels currently available on the IEEE 1394 bus.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
   ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            ULONG            fulSpeed;
            ULONG            BytesPerFrameAvailable;
            LARGE_INTEGER    ChannelsAvailable;
        } IsochQueryResources;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_QUERY_RESOURCES

u.IsochQueryResources.fulSpeed

Specifies the speed flag to use in allocating bandwidth. The possible speed values are SPEED_FLAGS_xxx, where xxx is the approximate transfer rate in megabits per second. Currently, existing hardware supports transfer rates of 100, 200, and 400 MBps.

IRB Output

u.IsochQueryResources.BytesPerFrameAvailable

On success, specifies the returned available bandwidth as expressed in bytes per isochronous frame.

u.IsochQueryResources.ChannelsAvailable

On success, points to a bitmap of available channels.

I/O Status Block

On success, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS.

Operation

The information returned by this request should not be relied upon; a node on the bus may request or release channels or bandwidth at any time.

Drivers that submit this request must be running at an IRQL of PASSIVE_LEVEL.

REQUEST_ISOCH_SET_CHANNEL_BANDWIDTH

The REQUEST_ISOCH_SET_CHANNEL_BANDWIDTH request resets the bandwidth on an already-allocated bandwidth handle.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
   ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            HANDLE    hBandwidth;
            ULONG     nMaxBytesPerFrame;
        } IsochSetChannelBandwidth;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_SET_CHANNEL_BANDWIDTH

u.IsochSetChannelBandwidth.hBandwidth

Bandwidth handle to reset.

u.IsochSetChannelBandwidth.nMaxBytesPerFrame

Specifies the new bandwidth for **hBandwidth**.

I/O Status Block

If the bus driver successfully resets the bandwidth on **hBandwidth**, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS.

Operation

This request does not require the caller to know the bandwidth that was allocated when a handle was generated. REQUEST_ISOCH_SET_CHANNEL_BANDWIDTH can be used to readjust the bandwidth on a bandwidth handle whose bytes per frame setting is unknown. Despite its name, this request does not involve isochronous channels in any way.

Drivers that submit this request must be running at an IRQL of PASSIVE_LEVEL.

REQUEST_ISOCH_STOP

The **IsochStop** request stops all isochronous operations on an isochronous channel.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            HANDLE    hResource;
            ULONG     fulFlags;
        } IsochStop;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_STOP

u.IsochStop.hResource

Specifies the resource handle for the channel on which to stop isochronous operations.

u.IsochStop.fulFlags

Reserved. Device drivers should set this member to zero.

I/O Status Block

On success, the bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS.

REQUEST_ISOCH_TALK

The REQUEST_ISOCH_TALK request begins data transfer on a isochronous channel.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
      .
      .
      .
    union {
        struct {
            HANDLE        hResource;
            ULONG         fulFlags;
            CYCLE_TIME    StartTime;
         } IsochTalk;
         .
         .
         .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_ISOCH_TALK

u.IsochTalk.hResource

Specifies the resource handle to use for this operation. Resources are acquired through the REQUEST_ISOCH_ALLOCATE_RESOURCES request.

u.IsochTalk.fulFlags

Reserved. Drivers should set this to zero.

u.IsochTalk.StartTime

Specifies the cycle time to begin operation. This member is used only if the driver specified the RESOURCE_SYNCH_ON_TIME flag when it allocated the resource handle passed in **u.IsochTalk.hResource**. (The timing resolution is per isochronous cycle, so the **CycleOffset** member of **StartTime** is not used.)

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS when the listening operation has begun, or to the appropriate error status.

Operation

Talking on a channel may begin immediately, or it may be synchronized to an event. If the driver set the RESOURCE_SYNCH_ON_TIME flag on the REQUEST_ISOCH_ALLOCATE_RESOURCES request that returned the resource handle, then the write operation will begin on the specified cycle count. Additional synchronization options can be set for each buffer in the ISOCH_DESCRIPTOR structure.

If successful, the request returns a STATUS_SUCCESS value. The call returns immediately, and does not wait for any synchronization events. The bus driver calls the callback the driver provides in ISOCH_DESCRIPTOR to signal that it has finished processing an attached buffer.

REQUEST_SEND_PHY_CONFIG_PACKET

The REQUEST_SEND_PHY_CONFIG_PACKET request sends a Phy configuration packet to the bus. A raw packet is sent out, with no checks performed.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            PHY_CONFIGURATION_PACKET    PhyConfigurationPacket;
        } SendPhyConfigurationPacket;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_SEND_PHY_CONFIG_PACKET.

u.SendPhyConfigurationPacket.PhyConfigurationPacket

Pointer to the PHY_CONFIGURATION_PACKET structure.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

Operation

The packet is sent to all nodes on the bus. See the IEEE 1394 Specification for a description of Phy packets.

REQUEST_SET_DEVICE_XMIT_PROPERTIES

The REQUEST_SET_DEVICE_XMIT_PROPERTIES request sets the maximum speed that is used to transmit requests to a particular device.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            ULONG    fulSpeed;
            ULONG    fulPriority;
        } SetDeviceXmitProperties;
        .
        .
        .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_SET_DEVICE_XMIT_PROPERTIES.

u.SetDeviceXmitProperties.fulSpeed

Specifies the maximum speed for transactions to the device. The possible speed values are SPEED_FLAGS_xxx, where xxx is the (approximate) transfer rate in megabits per second. At the time of this writing, the existing hardware supports transfer rates of 100, 200, and 400 Mb/sec.

u.SetDeviceXmitProperties.fulPriority

Reserved.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

Operation

By default, the maximum permitted transmission speed is the physical maximum. A driver should use this request to lower the maximum permitted speed.

REQUEST_SET_LOCAL_HOST_PROPERTIES

The REQUEST_SET_LOCAL_HOST_PROPERTIES request sets properties of the local host.

The relevant members of the IRB for this request are:

```
typedef struct _IRB {
    ULONG FunctionNumber;
        .
        .
        .
    union {
        struct {
            ULONG    nLevel;
            PVOID    Information;
        } GetLocalHostInformation;
          .
          .
          .
    } u;
} IRB;
```

IRB Input

FunctionNumber

REQUEST_SET_LOCAL_HOST_PROPERTIES.

u.GetLocalHostInformation.nLevel

Specifies the request level. Currently the only supported option is SET_LOCAL_HOST_PROPERTIES_GAP_COUNT.

u.GetLocalHostInformation.Information

Pointer to arguments for the **u.GetLocalHostInformation.nLevel** option. For **u.GetLocalHostInformation.nLevel** equal to SET_LOCAL_HOST_PROPERTIES_GAP_COUNT, **u.GetLocalHostInformation.Information** must point to this structure:

```
struct _SET_LOCAL_HOST_PROPS2 {
    ULONG       GapCountLowerBound;
} SET_LOCAL_HOST_PROPS2, *PSET_LOCAL_HOST_PROPS2;
```

GapCountLowerBound sets a lower bound on the value the bus will use for its gap count. See the *IEEE 1394-1995 Specification* for a description of the gap count.

I/O Status Block

The bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS on success, or the appropriate error code on failure.

CHAPTER 2

IEEE 1394 Structures

This chapter describes structures used by IEEE 1394 device drivers.

To use these structures, include the 1394.h header file that is shipped with the Windows® 2000 DDK.

ADDRESS_FIFO

```
typedef struct _ADDRESS_FIFO {
    SINGLE_LIST_ENTRY   FifoList;       // Singly linked list
    PMDL                FifoMdl;        // Mdl for this FIFO element
} ADDRESS_FIFO, *PADDRESS_FIFO;
```

The ADDRESS_FIFO structure is an entry in a singly-linked list of MDLs used by the REQUEST_ALLOCATE_ADDRESS_RANGE IEEE 1394 bus request.

Members

FifoList

Specifies the rest of the list.

FifoMdl

Specifies the MDL for this entry of the list.

ADDRESS_OFFSET

```
typedef struct _ADDRESS_OFFSET {
    USHORT        Off_High;
    ULONG         Off_Low;
} ADDRESS_OFFSET, *PADDRESS_OFFSET;
```

The ADDRESS_OFFSET structure specifies the 48-bit address within a device's IEEE 1394 address space.

Members

Off_High

Specifies the high order offset for a IEEE 1394 address.

Off_Low

Specifies the low order offset for a IEEE 1394 address.

ADDRESS_RANGE

```
typedef struct _ADDRESS_RANGE {
    USHORT      AR_Off_High;
    USHORT      AR_Length;
    ULONG       AR_Off_Low;
} ADDRESS_RANGE, *PADDRESS_RANGE;
```

The ADDRESS_RANGE structure describes a range in a IEEE 1394 device's address space.

Members

AR_Offset_High

Specifies the high order bits of the 1394 address within the buffer.

AR_Length

Specifies the length, in bytes, of a 1394 address buffer.

AR_Offset_Low

Specifies the low order bits of the 1394 address within the buffer.

CONFIG_ROM

```
typedef struct _CONFIG_ROM
{
    ULONG    CR_Info;
    ULONG    CR_Signiture;
    ULONG    CR_BusInfoBlockCaps;
    ULONG    CR_Node_UniqueID[2];
    ULONG    CR_Root_Info;

    // Other information may be provided by vendor
} CONFIG_ROM, *PCONFIG_ROM;
```

The CONFIG_ROM structure is used to contain the first 24 bytes of a IEEE 1394 device's configuration ROM.

Members

CR_Info

Specifies the first 4 bytes of the configuration ROM.

CR_Signiture

Specifies a signature that will be the same for all 1394 devices.

CR_BusInfoBlockCaps

Specifies the bus capabilities of the device.

CR_Node_UniqueID

Specifies the node's 64-bit vendor-assigned unique ID.

CR_Root_Info

Specifies the first 4 bytes of the root directory information.

Context

See the *IEEE 1394-1995 Specification* for more details about the layout of the standard configuration ROM.

CYCLE_TIME

```
typedef struct _CYCLE_TIME {
    ULONG     CL_CycleOffset:12;
    ULONG     CL_CycleCount:13;
    ULONG     CL_SecondCount:7;
} CYCLE_TIME, *PCYCLE_TIME;
```

The CYCLE_TIME structure contains the IEEE 1394 isochronous cycle time.

Members

CycleOffset

Specifies the number of clock ticks (based on a 24.576 MHz clock) since the current isochronous cycle began.

CycleCount

Specifies the number of isochronous cycles in the current second.

SecondCount

Specifies the number of seconds. This count wraps to zero every 128 seconds.

Comments

The layout of this structure matches that of the CYCLE_TIME register in the *IEEE 1394-1995 Specification.*

GET_LOCAL_HOST_INFO1

```
typedef struct _GET_LOCAL_HOST_INFO1 {
    LARGE_INTEGER       UniqueId;
} GET_LOCAL_HOST_INFO1, *PGET_LOCAL_HOST_INFO1;
```

The GET_LOCAL_HOST_INFO1 structure contains the data returned by a REQUEST_GET_LOCAL_HOST_INFO request using **u.GetLocalHostInformation.nLevel** GET_HOST_UNIQUE_ID.

Members

UniqueId

The bus driver fills in this member with the IEEE 1394 globally unique device ID for the host controller.

GET_LOCAL_HOST_INFO2

```
typedef struct _GET_LOCAL_HOST_INFO2 {
    ULONG               HostCapabilities;
    ULONG               MaxAsyncReadRequest;
    ULONG               MaxAsyncWriteRequest;
} GET_LOCAL_HOST_INFO2, *PGET_LOCAL_HOST_INFO2;
```

The GET_LOCAL_HOST_INFO2 structure contains the data returned by a REQUEST_GET_LOCAL_HOST_INFO request using **u.GetLocalHostInformation.nLevel** GET_HOST_CAPABILITIES.

Members

HostCapabilities

Flag	Description
HOST_INFO_PACKET_BASED	The host controller supports packet-based isochronous transactions.
HOST_INFO_STREAM_BASED	The host controller supports stream-based isochronous transactions.
HOST_INFO_SUPPORTS_ISOCH_STRIPPING	The host controller supports configurable stripping of header information.

Flag	Description
HOST_INFO_SUPPORTS_START_ON CYCLE	The host controller supports synchronizing start on specific isochronous cycle times.
HOST_INFO_SUPPORTS_RETURNING_ ISO_HDR	The host controller does not automatically strip off the isochronous packet header.
HOST_INFO_SUPPORTS_ISO_HDR_ INSERTION	The host controller supports the DESCRIPTOR_ HEADER_SCATTER_GATHER flag in ISOCH_ DESCRIPTOR.

MaxAsyncReadRequest

The bus driver fills in this member with the maximum size asynchronous read request that the host controller supports.

MaxAsyncWriteRequest

The bus driver fills in this member with the maximum size asynchronous write request that the host controller supports.

Comments

Port drivers that return the HOST_INFO_PACKET_BASED flag support the Isoch-AllocateResources request's RESOURCE_USE_PACKET_BASED flag. Port drivers that return the HOST_STREAM_PACKET_BASED flag support stream-based I/O. If the host controller does not support stream-based I/O, the driver must use packet-based I/O, and set the RESOURCE_USE_PACKET_BASED flag when issuing the REQUEST_ISOCH_ ALLOCATE_RESOURCES.

Some host controllers automatically strip the packet header off an isochronous packet. If a host controller does not automatically strip the header, the port driver will return the HOST_INFO_SUPPORTS_ISO_HDR flag. Some host controllers allow a driver to configure the host controller to strip off a given number of quadlets; for such controllers the port driver will return the HOST_INFO_SUPPORTS_ISOCH_STRIPPING flag. The driver should check this flag before calling REQUEST_ISOCH_ALLOCATE_RESOURCES with the RESOURCE_STRIP_ADDITIONAL_QUADLETS flag.

If a host controller supports synchronizing REQUEST_ISOCH_LISTEN and REQUEST_ ISOCH_TALK requests to the isochronous cycle time, the port driver will return the HOST_INFO_SUPPORTS_START_ON_CYCLE. Drivers should check this flag before attempting to issue an REQUEST_ISOCH_ALLOCATE_RESOURCES request using the RESOURCE_SYNCH_ON_TIME flag, or before attaching a buffer whose ISOCH_ DESCRIPTOR has the DESCRIPTOR_SYNCH_ON_TIME flag set.

GET_LOCAL_HOST_INFO3

```
typedef struct _GET_LOCAL_HOST_INFO3 {
    ULONG               deciWattsSupplied;
    ULONG               Voltage;        // x10 -> +3.3 == 33
                                        // +5.0 == 50,+12.0 == 120
                                        // etc.
} GET_LOCAL_HOST_INFO3, *PGET_LOCAL_HOST_INFO3;
```

The GET_LOCAL_HOST_INFO3 structure contains the data returned by a REQUEST_GET_LOCAL_HOST_INFO request using **u.GetLocalHostInformation.nLevel** = GET_POWER_SUPPLIED.

Members

deciWattsSupplied

Specifies the wattage supplied, in tenths of a watt.

Voltage

Specifies the voltage, in tenths of a volt.

GET_LOCAL_HOST_INFO4

```
typedef struct _GET_LOCAL_HOST_INFO4 {
    PPORT_PHYS_ADDR_ROUTINE PhysAddrMappingRoutine;
    PVOID                   Context;
} GET_LOCAL_HOST_INFO4, *PGET_LOCAL_HOST_INFO4;
```

The GET_LOCAL_HOST_INFO4 structure contains the data returned by a REQUEST_GET_LOCAL_HOST_INFO request using **u.GetLocalHostInformation.nLevel** = GET_PHYS_ADDR_ROUTINE.

Members

PhysAddrMappingRoutine

Pointer to the physical address mapping routine, which is of type:

```
NTSTATUS
PhysAddrMappingRoutine (
    IN PVOID Context,
    IN OUT PIRB Irb
);
```

The physical mapping routine is invoked on an REQUEST_ALLOCATE_ADDRESS_RANGE IRB. It fills in the **u.AllocateAddressRange.pAddressRange** member with the

physical addresses that the **u.AllocateAddressRange.Mdl** member of the IRB are mapped to. The proper value for the Context parameter is the **Context** member below.

Context

Specifies the argument that should be passed as the Context argument of the physical address mapping routine.

GET_LOCAL_HOST_INFO5

```
typedef struct _GET_LOCAL_HOST_INFO5 {
    PVOID               ConfigRom;
    ULONG               ConfigRomLength;
} GET_LOCAL_HOST_INFO5, *PGET_LOCAL_HOST_INFO5;
```

The GET_LOCAL_HOST_INFO5 structure contains the data returned by a REQUEST_GET_LOCAL_HOST_INFO request using **u.GetLocalHostInformation.nLevel** = GET_HOST_CONFIG_ROM.

Members

ConfigRom

Pointer to the beginning of the buffer to be filled with the local host's configuration ROM.

ConfigRomLength

Specifies the length of the buffer pointed to by **ConfigRom**.

Comments

When submitted in a REQUEST_GET_LOCAL_HOST_INFO request, if the **ConfigRomLength** is smaller than the size of the Configuration ROM, a status code of STATUS_INVALID_BUFFER_SIZE is returned. In this case, the correct buffer size will be filled in the **ConfigRomLength** member.

GET_LOCAL_HOST_INFO6

```
typedef struct _GET_LOCAL_HOST_INFO6 {
    ADDRESS_OFFSET          CsrBaseAddress;
    ULONG                   CsrDataLength;
    PVOID                   CsrDataBuffer;
} GET_LOCAL_HOST_INFO6, *PGET_LOCAL_HOST_INFO6;
```

The GET_LOCAL_HOST_INFO6 structure contains the data returned by a REQUEST_GET_LOCAL_HOST_INFO request using **u.GetLocalHostInformation.nLevel** = GET_HOST_CSR_CONTENTS.

Members

CsrBaseAddress

Specifies the base address to examine in the CSR. **CsrBaseAddress.Off_High** must be INITIAL_REGISTER_SPACE_HI. The possible values of **CsrBaseAddress.Off_Low** are:

CsrBaseAddress.Off_Low	Type of Data
SPEED_MAP_LOCATION	The current speed map. The bus driver converts this from big-endian to machine-native format before it returns the data.
TOPOLOGY_MAP_LOCATION	The current topology map. The bus driver converts this from big-endian to machine-native format before it returns the data.

CsrDataLength

Specifies the length in bytes of the buffer that **CsrDataBuffer** points to.

CsrDataBuffer

Pointer to the buffer where the bus driver returns the requested CSR data.

Comments

When submitted in a REQUEST_GET_LOCAL_HOST_INFO request, if the **CsrDataLength** is smaller than the size of the requested data, STATUS_INVALID_BUFFER_SIZE is returned. In this case, the correct buffer size will be filled in the **CsrDataLength** member.

IO_ADDRESS

```
typedef struct _IO_ADDRESS {
    NODE_ADDRESS      IA_Destination_ID;
    ADDRESS_OFFSET    IA_Destination_Offset;
} IO_ADDRESS, *PIO_ADDRESS;
```

The IO_ADDRESS structure specifies the 1394 64-bit destination address for read, write and lock operations.

Members

IA_Destination_ID

Pointer to the destination node address.

IA_Destination_Offset

Specifies the index of the 1394 address within the address array.

IRB

```
typedef struct _IRB {
    ULONG FunctionNumber;
    ULONG Flags;
    ULONG BusReserved[IRB_BUS_RESERVED_SZ];
    ULONG PortReserved[IRB_PORT_RESERVED_SZ];
    union {
        .
        .
        .
    } u;
}
```

Drivers use this structure to pass requests to IEEE 1394 bus driver.

Members

FunctionNumber

Determines the type of request. Each request type is documented under the value of **FunctionNumber** in *IEEE 1394 Bus I/O requests.*

Flags

Reserved. Drivers must set this member to zero.

BusReserved

Reserved.

PortReserved

Reserved.

u

Specifies a union of structures, one for each value of **FunctionNumber**. The applicable submembers of u for each request are described with each request type inChapter 1, *IEEE 1394 Bus I/O Requests.*

FunctionNumber	Associated member
REQUEST_ALLOCATE_ADDRESS_RANGE	AllocateAddressRange
REQUEST_ASYNC_LOCK	AsyncLock
REQUEST_ASYNC_READ	AsyncRead
REQUEST_ASYNC_STREAM	AsyncStream
REQUEST_ASYNC_WRITE	AsyncWrite
REQUEST_BUS_RESET	BusReset

Continued

FunctionNumber	Associated member
REQUEST_BUS_RESET_NOTIFICATION	BusResetNotification
REQUEST_CONTROL	Control
REQUEST_ISOCH_ALLOCATE_BANDWIDTH	IsochAllocateBandwidth
REQUEST_ISOCH_ALLOCATE_CHANNEL	IsochAllocateChannel
REQUEST_ISOCH_ALLOCATE_RESOURCES	IsochAllocateResources
REQUEST_ISOCH_ATTACH_BUFFERS	IsochAttachBuffers
REQUEST_ISOCH_DETACH_BUFFERS	IsochDetachBuffers
REQUEST_ISOCH_FREE_BANDWIDTH	IsochFreeBandwidth
REQUEST_ISOCH_FREE_CHANNEL	IsochFreeChannel
REQUEST_ISOCH_FREE_RESOURCES	IsochFreeResources
REQUEST_ISOCH_LISTEN	IsochListen
REQUEST_ISOCH_QUERY_CYCLE_TIME	IsochQueryCurrentCycleTime
REQUEST_ISOCH_QUERY_RESOURCES	IsochQueryResources
REQUEST_ISOCH_SET_CHANNEL_BANDWIDTH	IsochSetChannelBandwidth
REQUEST_ISOCH_STOP	IsochStop
REQUEST_ISOCH_TALK	IsochTalk
REQUEST_FREE_ADDRESS_RANGE	FreeAddressRange
REQUEST_GET_ADDR_FROM_DEVICE_OBJECT	Get1394AddressFromDeviceObject
REQUEST_GET_CONFIGURATION_INFO	GetConfigurationInformation
REQUEST_GET_GENERATION_COUNT	GetGenerationCount
REQUEST_GET_LOCAL_HOST_INFO	GetLocalHostInformation
REQUEST_GET_SPEED_BETWEEN_DEVICES	GetMaxSpeedBetweenDevices
REQUEST_GET_SPEED_TOPOLOGY_MAPS	GetSpeedTopologyMaps
REQUEST_SEND_PHY_CONFIG_PACKET	SendPhyConfigurationPacket
REQUEST_SET_LOCAL_HOST_PROPERTIES	SetLocalHostProperties
REQUEST_SET_DEVICE_XMIT_PROPERTIES	SetDeviceXmitProperties

Comments

The **Parameters->Others.Arguments1** member of an IOCTL_CLASS_1394 IRP points to an IRB structure. The bus driver uses the IRB to determine the type of request made by the device driver, and also to return the results of the operation. See Chapter 1, *IEEE 1394 Bus I/O Requests* for a description of the behavior of each request.

ISOCH_DESCRIPTOR

```
typedef struct _ISOCH_DESCRIPTOR {
    ULONG       fulFlags;
    PMDL        Mdl;
    ULONG       ulLength;
    ULONG       nMaxBytesPerFrame;
    ULONG       ulSynch;
    ULONG       ulTag;
    CYCLE_TIME  CycleTime;
    PBUS_ISOCH_DESCRIPTOR_ROUTINE   Callback;
    PVOID       Context1;
    PVOID       Context2;
    NTSTATUS    status;
    ULONG       DeviceReserved[8];
    ULONG       BusReserved[8];
    ULONG       PortReserved[16];
} ISOCH_DESCRIPTOR, *PISOCH_DESCRIPTOR;
```

The ISOCH_DESCRIPTOR structure describes a buffer to be attached or detailed from a resource handle, using the REQUEST_ISOCH_ATTACH_BUFFERS and REQUEST_ISOCH_DETACH_BUFFERS requests.

Members

fulFlags

Specifies various flags for this isochronous descriptor:

Flag	Isochronous Transaction	Description
DESCRIPTOR_SYNCH_ON_SY	Listen	The host controller waits for a particular Sy value that is embedded in the isochronous packet header before continuing to read data. The Sy value is specified in **ulSynch**.
DESCRIPTOR_SYNCH_ON_TAG	Listen	The host controller waits for a particular Tag value that is embedded in the isochronous packet header before continuing to read data. The tag value is specified in **ulTag**.
DESCRIPTOR_SYNCH_ON_TIME	Listen, Talk	The host controller waits for a particular isochronous cycle time before continuing the operation. The cycle time is specified in the **CycleTime** member.
DESCRIPTOR_USE_SY_TAG_IN_FIRST	Listen	Synchronization on the **Sy** or **Tag** members occurs only for the first packet received.

Continued

Flag	Isochronous Transaction	Description
DESCRIPTOR_TIME_STAMP_ON_COMPLETION	Listen, Talk	Once the host controller completes its DMA to or from this buffer, store the cycle time in the **CycleTime** member of the ISOCH_DESCRIPTOR.
DESCRIPTOR_PRIORITY_TIME_DELIVERY	Talk	If the local host controller is not ready for a write, do not retry the write later. (The default behavior is to retry until the host controller is ready.)
DESCRIPTOR_HEADER_SCATTER_GATHER	Talk	The host controller treats the data in this buffer as a sequence of headers. The host controller will prepend a header from this buffer to each packet it assembles from the data in the next buffer attached.

Mdl

Specifies the MDL representing a buffer in which the data is, or will be, contained.

ulLength

Specifies the length of the **Mdl**.

nMaxBytesPerFrame

Specifies the maximum bytes contained in each isochronous frame. On writes, the data in the buffer will be split into isochronous packets of this size.

ulSynch

For IsochTalk requests, this member specifies the Sy field of the outgoing packet. For REQUEST_ISOCH_LISTEN requests, if the DESCIPTOR_SYNCH_ON_SY flag is set, this member specifies the value the host controller will match against the Sy field in isochronous packet headers.

ulTag

For IsochTalk requests, this member specifies the Tag field of the outgoing packet. For REQUEST_ISOCH_LISTEN requests, if the DESCIPTOR_SYNCH_ON_SY flag is set, this member specifies the value the host controller will match against the Tag field in isochronous packet headers.

CycleTime

If the DESCRIPTOR_SYNCH_ON_TIME flag is set, this member specifies the isochronous cycle time to synchronize on. (The timing resolution is per isochronous cycle. The **CycleOffset** member of the cycle time is not used.) If the DESCRIPTOR_TIME_STAMP_ON_COMPLETION flag is set, the bus driver will fill this member with the isochronous cycle time on completion of the operation that used this buffer.

Callback

Pointer to a callback routine. If non-NULL, the bus driver calls this routine when DMA to the buffer has completed. The callback executes at IRQL DISPATCH_LEVEL. The callback is of the following type:

```
void Callback(IN PVOID Context1, IN PVOID Context2);
```

Context1

Specifies the first parameter when the bus driver calls the routine passed in **Callback**.

Context2

Specifies the second parameter when the bus driver calls the routine passed in **Callback**.

status

Specifies the final status of the operation on this buffer. The bus driver fills in **status** when DMA to or from this buffer is finished.

DeviceReserved

Reserved.

BusReserved

Reserved.

PortReserved

Reserved.

Comments

Not all DESCRIPTOR_XXX flags are supported on all hardware. The device driver can use the REQUEST_GET_LOCAL_HOST_INFO request, with **nLevel** = GET_HOST_CAPABILITIES, to determine which DESCRIPTOR_XXX flags are supported. The bus driver returns a pointer to a GET_LOCAL_HOST_INFO2 structure, whose **HostCapabilities** member contains flags that determine which flags the host controller supports. The following table lists which DESCRIPTOR_XXX flags require hardware support, and the corresponding **HostCapabilities** flag the driver should check:

DESCRIPTOR_XXX flags	HostCapabilities
DESCRIPTOR_SYNCH_ON_TIME	HOST_INFO_SUPPORTS_START_ON_CYCLE
DESCRIPTOR_HEADER_SCATTER_GATHER	HOST_INFO_SUPPORTS_ISO_HDR_INSERTION

If the driver sets the DESCRIPTOR_HEADER_SCATTER_GATHER flag, the host controller will combine the data of the buffer specified in **Mdl** with the data of the next buffer attached. (Subsequent buffers are unaffected.) Each frame of the buffer will be prepended to a frame of the next buffer (in the order the data in the buffer is split into frames), and sent as the data of the next isochronous packet. The number of frames of each buffer must match, or the bus driver returns STATUS_INVALID_PARAMETER for the next REQUEST_ISOCH_ATTACH_BUFFER request.

The DESCRIPTOR_HEADER_SCATTER_GATHER flag is not supported on Windows 98. It is supported on Windows 2000, and later.

NODE_ADDRESS

```
typedef struct _NODE_ADDRESS {
    USHORT      NA_Node_Number:6;    // Bits 10-15
    USHORT      NA_Bus_Number:10;    // Bits 0-9
} NODE_ADDRESS, *PNODE_ADDRESS;
```

The NODE_ADDRESS structure specifies the 10-bit bus number and 6-bit node number that serve as the node address for a 1394 node.

Members

NA_Node_Number

Specifies the 6-bit node number.

NA_Bus_Number

Specifies the 10-bit bus number.

NOTIFICATION_INFO

```
typedef struct _NOTIFICATION_INFO {
    PMDL                Mdl;
    ULONG               ulOffset;
    ULONG               nLength;
    ULONG               fulNotificationOptions;
    PVOID               Context;
    PADDRESS_FIFO       Fifo;
    PVOID               RequestPacket;
    PMDL                ResponseMdl;
    PVOID *             ResponsePacket;
    PULONG              ResponseLength;
    PKEVENT *           ResponseEvent;
} NOTIFICATION_INFO, *PNOTIFICATION_INFO;
```

The bus driver passes NOTIFICATION_INFO to pass information to the driver-provided notification routine for a driver-allocated address range in the computer's IEEE 1394 address space. The bus driver calls the notification routine when it receives an asynchronous I/O request packet for that address.

Members

Mdl

If non-NULL, **Mdl** specifies the MDL for the allocated address range.

ulOffset

Specifies the byte offset with the MDL that corresponds to the address that received a request packet. Only used when **Mdl** is non-NULL.

nLength

Specifies the number of bytes affected by the request packet. Only used when **Mdl** is non-NULL.

fulNotificationOptions

Specifies which type of event triggered the bus driver to call the notification routine. The possible events the bus driver can return are: NOTIFY_FLAGS_AFTER_READ, NOTIFY_FLAGS_AFTER_WRITE, or NOTIFY_FLAGS_AFTER_LOCK. Only used when **Mdl** is non-NULL.

Context

Pointer to specific context data for this allocated address range. The driver supplies this data through the **u.AllocateAddressRange.Context** member of the IRB for the original REQUEST_ALLOCATE_ADDRESS_RANGE request.

Fifo

Pointer to the ADDRESS_FIFO structure containing the FIFO element just completed. Only used if the driver submitted an ADDRESS_FIFO list in the original REQUEST_ALLOCATE_ADDRESS_RANGE request.

RequestPacket

If non-NULL, **RequestPacket** points to the original request packet. The bus driver only supplies this if the device driver did not supply an MDL or an ADDRESS_FIFO list in the original REQUEST_ALLOCATE_ADDRESS_RANGE request.

ResponseMdl

If non-NULL, **ResponseMdl** points to an uninitialized MDL. The driver must initialize this MDL for a non-pageable buffer, and fill the buffer with the response packet. The bus driver

only supplies this if the device driver did not supply an MDL or an ADDRESS_FIFO list in the original REQUEST_ALLOCATE_ADDRESS_RANGE request.

ResponsePacket

If non-NULL, **ResponsePacket** points to a memory location that the driver fills in with a pointer to the beginning of its response packet. The bus driver only supplies this if the device driver did not supply an MDL or an ADDRESS_FIFO list in the original REQUEST_ALLOCATE_ADDRESS_RANGE request.

ResponseLength

If non-NULL, **ResponseLength** points to a memory location that the driver fills in with the length of its response packet. The bus driver only supplies this if the device driver did not supply an MDL or an ADDRESS_FIFO list in the original REQUEST_ALLOCATE_ADDRESS_RANGE request.

ResponseEvent

If non-NULL, **ResponseEvent** points to a memory location that the driver fills in with the kernel event the bus driver should use to signal that it has completed sending the response packet. The bus driver only supplies this if the device driver did not supply an MDL or an ADDRESS_FIFO list in the original REQUEST_ALLOCATE_ADDRESS_RANGE request.

Comments

When a driver allocates an address range on the computer's IEEE 1394 address space, it may require the bus driver to notify it for some or all request packets sent to the allocated addresses. As part of the original allocate request, the driver may either require the bus driver to forward each packet to the driver for handling, or it may require the bus driver to handle the packet and merely notify the device driver when it has finished. See REQUEST_ALLOCATE_ADDRESS_RANGE in Chapter 1 for details.

If the device driver provides no backing store, the bus driver forwards each packet to the device driver to handle. The bus driver passes NULL for **Mdl**, and passes the packet in **RequestPacket**. The bus driver also passes pointers to memory locations that the device driver must fill in with the buffer for the response packet (in **ResponsePacket**), the buffer length (in **ResponseLength**), and an MDL for the buffer (in **ResponseMdl**). The bus driver also supplies a memory location the driver can use to pass a kernel event object in **ResponseEvent**. If the device driver provides an event object, the bus driver will use it to signal the driver when it has finished sending the response packet.

The request packet is in whatever form returned by the host controller, and the response packet must be in the same form. This makes it difficult to write portable code that does not depend on the host controller.

If the driver provides backing store in the original allocate address range request, the bus driver uses the driver's notification routine to signal that it has completed reading or writing data from the backing store. It passes the MDL of the backing store in the **Mdl** member, and the starting location and size within the associated buffer in **ulOffset** and **nLength**. The bus driver also passes the type of event that led to notification in **fulNotificationOptions**.

If the device driver is using a linked list of ADDRESS_FIFO's as backing store, the bus driver returns the list element it popped off in **Fifo**.

PHY_CONFIGURATION_PACKET

```
typedef struct _PHY_CONFIGURATION_PACKET {
    ULONG               PCP_Phys_ID:6;          // Byte 0 - Bits 0-5
    ULONG               PCP_Packet_ID:2;        // Byte 0 - Bits 6-7
    ULONG               PCP_Gap_Count:6;        // Byte 1 - Bits 0-5
    ULONG               PCP_Set_Gap_Count:1;    // Byte 1 - Bit 6
    ULONG               PCP_Force_Root:1;       // Byte 1 - Bit 7
    ULONG               PCP_Reserved1:8;        // Byte 2 - Bits 0-7
    ULONG               PCP_Reserved2:8;        // Byte 3 - Bits 0-7
    ULONG               PCP_Inverse;            // Inverse quadlet
} PHY_CONFIGURATION_PACKET, *PPHY_CONFIGURATION_PACKET;
```

The PHY_CONFIGURATION_PACKET structure contains a raw PHY configuration packet. See the IEEE 1394 specification for details.

Members

PCP_Phys_ID

Specifies the node address of the root.

PCP_Packet_ID

This member must be PHY_PACKET_ID_CONFIGURATION to indicate it is a PHY configuration packet.

PCP_Gap_Count

If the **PCP_Set_Gap_Count** bit is set, the PHY register gap_count field is set to this value.

PCP_Set_Gap_Count

If this bit is set, the PHY register gap_count field is set to **PCP_Gap_Count**.

PCP_Force_Root

If set, the caller will become the root node.

PCP_Reserved1

Reserved.

PCP_Reserved2

Reserved.

PCP_Inverse

Specifies the logical inverse of the first quadlet of the packet.

SELF_ID

```
typedef struct _SELF_ID {
    ULONG               SID_Phys_ID:6;          // Byte 0 - Bits 0-5
    ULONG               SID_Packet_ID:2;        // Byte 0 - Bits 6-7
    ULONG               SID_Gap_Count:6;        // Byte 1 - Bits 0-5
    ULONG               SID_Link_Active:1;      // Byte 1 - Bit 6
    ULONG               SID_Zero:1;             // Byte 1 - Bit 7
    ULONG               SID_Power_Class:3;      // Byte 2 - Bits 0-2
    ULONG               SID_Contender:1;        // Byte 2 - Bit 3
    ULONG               SID_Delay:2;            // Byte 2 - Bits 4-5
    ULONG               SID_Speed:2;            // Byte 2 - Bits 6-7
    ULONG               SID_More_Packets:1;     // Byte 3 - Bit 0
    ULONG               SID_Initiated_Rst:1;    // Byte 3 - Bit 1
    ULONG               SID_Port3:2;            // Byte 3 - Bits 2-3
    ULONG               SID_Port2:2;            // Byte 3 - Bits 4-5
    ULONG               SID_Port1:2;            // Byte 3 - Bits 6-7
} SELF_ID, *PSELF_ID;
```

The SELF_ID structure contains a raw packet zero self-ID packet. See the IEEE 1394 specification for details.

SID_Phys_ID

Specifies the device node number.

SID_Packet_ID

Must be PHY_PACKET_ID_SELF_ID.

SID_Gap_Count

Specifies the current value of the node's PHY_CONFIGURATION register's gap_count member.

SID_Link_Active

One if the device's link and transaction layers are active, zero otherwise.

SID_Zero

Always zero.

SID_Power_Class

The possible power classes are:

POWER_CLASS_NOT_NEED_NOT_REPEAT
POWER_CLASS_SELF_POWER_PROVIDE_15W
POWER_CLASS_SELF_POWER_PROVIDE_30W
POWER_CLASS_SELF_POWER_PROVIDE_45W
POWER_CLASS_MAYBE_POWERED_UPTO_1W
POWER_CLASS_IS_POWERED_UPTO_1W_NEEDS_2W
POWER_CLASS_IS_POWERED_UPTO_1W_NEEDS_5W
POWER_CLASS_IS_POWERED_UPTO_1W_NEEDS_9W

SID_Contender

One if this node is a contender for bus or isochronous resource manager, zero otherwise.

SID_Delay

Currently always zero.

SID_More_Packets

One if this packet will be followed by SELF_ID_MORE packets, zero otherwise.

SID_Initiated_Rst

One if this node initiated the most recent bus reset, zero otherwise.

SID_Port1

SID_Port2

SID_Port3

Specifies port status. Possible values are:

SELF_ID_CONNECTED_TO_CHILD
SELF_ID_CONNECTED_TO_PARENT
SELF_ID_NOT_CONNECTED
SELF_ID_NOT_PRESENT

SELF_ID_MORE

```
typedef struct _SELF_ID_MORE {
    ULONG               SID_Phys_ID:6;          // Byte 0 - Bits 0-5
    ULONG               SID_Packet_ID:2;        // Byte 0 - Bits 6-7
    ULONG               SID_PortA:2;            // Byte 1 - Bits 0-1
    ULONG               SID_Reserved2:2;        // Byte 1 - Bits 2-3
    ULONG               SID_Sequence:2;         // Byte 1 - Bits 4-5
    ULONG               SID_One:1;              // Byte 1 - Bit 6
    ULONG               SID_Reserved1:1;        // Byte 1 - Bit 7
```

```
    ULONG               SID_PortE:2;            // Byte 2 - Bits 0-1
    ULONG               SID_PortD:2;            // Byte 2 - Bits 2-3
    ULONG               SID_PortC:2;            // Byte 2 - Bits 4-5
    ULONG               SID_PortB:2;            // Byte 2 - Bits 6-7
    ULONG               SID_More_Packets:1;     // Byte 3 - Bit 0
    ULONG               SID_Reserved3:1;        // Byte 3 - Bit 1
    ULONG               SID_PortH:2;            // Byte 3 - Bits 2-3
    ULONG               SID_PortG:2;            // Byte 3 - Bits 4-5
    ULONG               SID_PortF:2;            // Byte 3 - Bits 6-7
} SELF_ID_MORE, *PSELF_ID_MORE;
```

The SELF_ID_MORE structure contains a raw packet one, two, or three self-ID packet. See the *IEEE 1394 Specification* for details.

Members

SID_Phys_ID

Specifies the device node number.

SID_Packet_ID

Must be PHY_PACKET_ID_SELF_ID.

SID_PortA

SID_PortB

SID_PortC

SID_PortD

SID_PortE

SID_PortF

SID_PortG

SID_PortH

Specifies port status. Possible values are:

SELF_ID_CONNECTED_TO_CHILD
SELF_ID_CONNECTED_TO_PARENT
SELF_ID_NOT_CONNECTED
SELF_ID_NOT_PRESENT

SID_Reserved1

SID_Reserved2

SID_Reserved3

Reserved.

SID_Sequence

Specifies the packet number in sequence (the first SELF_ID_MORE packet is packet zero).

SID_One

Always a 1.

SID_More_Packets

One if this packet will be followed by more SELF_ID_MORE packets, zero otherwise.

SPEED_MAP

```
typedef struct _SPEED_MAP {
    USHORT              SPD_Length;
    USHORT              SPD_CRC;
    ULONG               SPD_Generation;
    UCHAR               SPD_Speed_Code[4032];
} SPEED_MAP, *PSPEED_MAP;
```

The SPEED_MAP structure is used to store a IEEE 1394 bus speed map. It describes the maximum speed obtainable by the devices on the bus.

Members

SPD_Length

Specifies the number of quadlets in the speed map.

SPD_CRC

Specifies the CRC value for the speed map.

SPD_Generation

Specifies the bus reset generation for which the speed map was created.

SPD_Speed_Code

Specifies an array of speed codes. Currently, the possible values are:

SCODE_100_RATE
SCODE_200_RATE
SCODE_400_RATE

Comments

All data will be in big-endian format.

TEXTUAL_LEAF

```
typedef struct _TEXTUAL_LEAF
{
    USHORT    TL_CRC;
    USHORT    TL_Length;
    ULONG     TL_Spec_Id;
    ULONG     TL_Language_Id;
    UCHAR     TL_Data;
} TEXTUAL_LEAF, *PTEXTUAL_LEAF;
```

The TEXTUAL_LEAF structure describes the device description that can be stored in the Configuration ROM of devices that satisfy the PC 98 or PC 99 specifications.

Members

TL_CRC

Specifies the CRC of the text string.

TL_Length

Specifies the length of the text string, in bytes.

TL_Spec_Id

Specifies which specification describes the meaning of the **TL_Language_ID** member.

TL_Language_Id

Specifies the language of the **TL_Data** member.

TL_Data

Specifies a vendor-specified textual description of the device.

TOPOLOGY_MAP

```
typedef struct _TOPOLOGY_MAP {
    USHORT      TOP_Length;
    USHORT      TOP_CRC;
    ULONG       TOP_Generation;
    USHORT      TOP_Node_Count;
    USHORT      TOP_Self_ID_Count;
    SELF_ID     TOP_Self_ID_Array[];
} TOPOLOGY_MAP, *PTOPOLOGY_MAP;
```

The TOPOLOGY_MAP structure is used to store an IEEE 1394 bus topology map. The relations between devices are found in the port members of the entries in **TOP_Self_ID_Array**.

Members

TOP_Length

Specifies the length in quadlets of the topology map.

TOP_CRC

Specifies the CRC value for the topology map.

TOP_Generation

Specifies the bus reset generation for which the topology map was created.

TOP_Node_Count

Specifies the number of nodes in the topology map.

TOP_Self_ID_Count

Specifies the number of entries in **TOP_Self_ID_Array**.

TOP_Self_ID_Array

Pointer to an array of SELF_ID and SELF_ID_MORE structures (the two structures are the same size).

Comments

All data will be in big-endian format.

PART 6

PCMCIA Drivers

CHAPTER 1

PCMCIA_INTERFACE_STANDARD Interface Memory Card Routines

This chapter describes the PCMCIA_INTERFACE_STANDARD interface routines provided by the Microsoft® Windows® 2000 system PCMCIA bus driver. PCMCIA memory card drivers can call these routines to perform the following operations:

- Modify the attributes of the memory window that is mapped by the PCMCIA bus driver
- Set the *Vpp* (secondary power source) level for the device
- Determine if the card memory is write-protected

To obtain pointers to these interface routines, a driver sends the PCMCIA bus driver an IRP_MJ_PNP request that specifies a IRP_MN_QUERY_INTERFACE minor function. The bus driver returns the interface information in a PCMCIA_INTERFACE_STANDARD structure:

```
typedef struct _PCMCIA_INTERFACE_STANDARD {
    USHORT    Size;
    USHORT    Version;
    PINTERFACE_REFERENCE    InterfaceReference;
    PINTERFACE_DEREFERENCE    InterfaceDereference;
    PVOID    Context;
    PPCMCIA_MODIFY_MEMORY_WINDOW    ModifyMemoryWindow;
    PPCMCIA_SET_VPP    SetVpp;
    PPCMCIA_IS_WRITE_PROTECTED    IsWriteProtected;
} PCMCIA_INTERFACE_STANDARD;
```

For more information on how to obtain a PCMCIA_INTERFACE_STANDARD interface, see *PCMICA_INTERFACE_STANDARD Interface for Memory Cards* in the *Kernel-Mode Drivers Design Guide* in the online DDK.

The routines in this chapter are listed in alphabetical order.

PCMCIA_IS_WRITE_PROTECTED

```
BOOLEAN
(*PPCMCIA_IS_WRITE_PROTECTED) (
  IN PVOID Context
  );
```

The **PCMCIA_IS_WRITE_PROTECTED** interface routine returns the write-protect condition of a PCMCIA memory card.

Parameters

Context

Pointer to the context for the interface routine.

Include

ntddpcm.h

Return Value

The **PCMCIA_IS_WRITE_PROTECTED** interface routine returns TRUE if the memory card is write-protected, otherwise it returns FALSE.

Comments

A caller must set the *Context* parameter to the context that is specified by the PCMCIA bus driver. The PCMCIA bus driver returns the context for the interface routines in the **Context** member of the same PCMCIA_INTERFACE_STANDARD structure that contains the pointers to the interface routines. If the *Context* parameter is not valid, system behavior is not defined, and the system might halt.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL. To maintain overall system performance, it is recommended that drivers call this routine at IRQL < DISPATCH_LEVEL.

See Also

PCMCIA_MODIFY_MEMORY_WINDOW, PCMCIA_SET_VPP

PCMCIA_MODIFY_MEMORY_WINDOW

```
BOOLEAN
(*PPCMCIA_MODIFY_MEMORY_WINDOW) (
  IN PVOID Context,
  IN PHYSICAL_MEMORY HostBase,
  IN PHYSICAL_MEMORY CardBase,
  IN BOOLEAN Enable,
  IN ULONG WindowSize,
  IN UCHAR AccessSpeed,
  IN UCHAR BusWidth,
  IN BOOLEAN AttributeMemory
  );
```

The **PCMCIA_MODIFY_MEMORY_WINDOW** interface routine sets the attributes of a memory window for a PCMCIA memory card. The memory window is mapped by the PCMCIA bus driver.

Parameters

Context

Pointer to the context for the interface routine.

HostBase

Specifies the physical memory window to map. *HostBase* is the base address for the memory card in the system's physical address space.

CardBase

Specifies the byte offset in the PCMCIA card's memory where the memory mapping begins.

Enable

Specifies permission to access the memory window. If *Enable* is TRUE, memory access is permitted, otherwise memory access is not permitted.

WindowSize

Specifies the size, in bytes, of the memory window that is mapped. The value of *WindowSize* cannot exceed the memory window granted to the driver in its assigned resources. If the value of Enable is TRUE and the value of WindowSize is zero, the size of the memory window granted to the driver in its assigned resources is used. If *Enable* is FALSE, *WindowSize* is not used.

AccessSpeed

Specifies the access speed of the PCMCIA card. The value of *AccessSpeed* is encoded as specified by the *PC Card Standard, Release 6.1*. If Enable is FALSE, *AccessSpeed* is not used.

BusWidth

Specifies the width of bus access to the PCMCIA memory card. *BusWidth* must be one of the following values:

PCMCIA_MEMORY_8BIT_ACCESS

PCMCIA_MEMORY_16BIT_ACCESS

If *Enable* is FALSE, *BusWidth* is not used.

AttributeMemory

Must be FALSE.

Include

ntddpcm.h

Return Value

The **PCMCIA_MODIFY_MEMORY_WINDOW** interface routine returns TRUE if the memory window is successfully enabled or disabled, as specified by the *Enable* parameter.

Comments

A caller must set the *Context* parameter to the context that is specified by the PCMCIA bus driver. The PCMCIA bus driver returns the context for the interface routines in the **Context** member of the same PCMCIA_INTERFACE_STANDARD structure that contains the pointers to the interface routines. If the *Context* parameter is not valid, system behavior is not defined, and the system might halt.

Callers of this routine must be running at IRQL <= DISPATCH_LEVEL. To maintain overall system performance, it is recommended that drivers call this routine at IRQL < DISPATCH_LEVEL.

See Also

PCMCIA_IS_WRITE_PROTECTED, **PCMCIA_SET_VPP**, PCMCIA_INTERFACE_STANDARD

PCMCIA_SET_VPP

```
BOOLEAN
  (*PPCMCIA_SET_VPP) (
  IN PVOID Context,
  IN UCHAR VppLevel
  );
```

The **PCMCIA_SET_VPP** interface routine sets the power level of the Vpp PCMCIA pin (secondary power source).

Parameters

Context

Pointer to the context for the interface routine.

VppLevel

Specifies the voltage level to set on the Vpp pin. *VppLevel* must be one of the following values:

PCMCIA_VPP_0V

Specifies that the voltage on the Vpp pin be set to zero volts and that the Vpp pin be disabled.

PCMCIA_VPP_12V

Specifies that the voltage on the Vpp pin be set to twelve volts.

PCMCIA_VPP_IS_VCC

Specifies that the voltage on the Vpp pin be set to equal the voltage on the Vcc (primary card power) pin.

Include

ntddpcm.h

Return Value

The **PCMCIA_SET_VPP** interface routine returns TRUE after the requested voltage level is set.

Comments

The **PCMCIA_SET_VPP** interface routine returns control to the caller after the requested voltage is established in a stable state for the card.

A caller must set the *Context* parameter to the context that is specified by the PCMCIA bus driver. The PCMCIA bus driver returns the context for the interface routines in the **Context** member of the same PCMCIA_INTERFACE_STANDARD structure that contains the pointers to the interface routines. If the *Context* parameter is not valid, system behavior is not defined, and the system might halt.

Callers of this routine can run at IRQL <= DISPATCH_LEVEL. To maintain overall system performance, it is recommended that drivers call this routine at IRQL < DISPATCH_LEVEL.

See Also

PCMCIA_IS_WRITE_PROTECTED, PCMCIA_MODIFY_MEMORY_WINDOW

PART 7

SMB Client Drivers

CHAPTER 1

SMB IOCTLS

This chapter describes the internal I/O control codes defined for Windows 2000 System Management Bus (SMB) drivers. Windows 98 does not provide support for SMB drivers.

An SMB miniport driver or a client of such a driver can send IRP_MJ_INTERNAL_DEVICE_CONTROL requests that specify the following I/O control codes:

SMB_BUS_REQUEST
SMB_REGISTER_ALARM_NOTIFY
SMB_DEREGISTER_ALARM_NOTIFY

When the SMB class driver receives the IRP that contains the IOCTL, it either performs the requested action or calls the miniport driver to perform the action.

Drivers that send or handle IRPs that contain these IOCTLs must include the header file *Smb.h*.

SMB_BUS_REQUEST

Operation

The class driver performs the requested action. If the device is idle, the class driver starts I/O on the device. Otherwise, the driver puts the request in the device I/O queue, where it might be handled by the miniport driver.

Input

Irp->Parameters.DeviceIoControl.Type3InputBuffer points to an SMB_REQUEST structure that describes the I/O request.

Irp->Parameters.DeviceIoControl.InputBufferLength specifies the length of the SMB_REQUEST structure.

Output

None.

I/O Status Block

IoStatus.Status is set as follows:

Set to	If
STATUS_SUCCESS	The operation succeeded.
STATUS_PENDING	The operation has been queued.
STATUS_BUFFER_TOO_SMALL or STATUS_INVALID_PARAMETER.	An error occurred.

IoStatus.Information is set to the size of the returned SMB_REQUEST structure when the IRP completes. If the IRP is pending, **Information** is set to zero. If the input buffer is too small, **Information** is set to the required length of the input buffer.

See Also

SMB_REQUEST

SMB_DEREGISTER_ALARM_NOTIFY

Operation

The SMB class driver deletes the requested alarm notification.

Input

Parameters.DeviceIoControl.Type3InputBuffer points to a handle previously returned by SMB_REGISTER_ALARM_NOTIFY.

Parameters.DeviceIoControl.InputBufferLength specifies the size of the handle.

Output

None.

I/O Status Block

IoStatus.Status is set to STATUS_SUCCESS if the operation succeeded, or to an error status such as STATUS_INVALID_PARAMETER otherwise.

See Also

SMB_REGISTER_ALARM_NOTIFY

SMB_REGISTER_ALARM_NOTIFY

Operation

The SMB class driver registers an alarm notification function for alarms that occur within a specified range of bus addresses.

When the miniport driver calls **SmbClassAlarm** to notify the SMB class driver that a device has signaled an alarm, the class driver calls the notification function.

Input

Parameters.DeviceIoControl.Type3InputBuffer points to an SMB_REGISTER_ALARM structure.

Parameters.DeviceIoControl.InputBufferLength specifies the length of the SMB_ REGISTER_ALARM structure.

Parameters.DeviceIoControl.OutputBufferLength specifies the number of bytes allocated for the returned handle (**sizeof**(PVOID)).

Output

Irp->UserBuffer points to a handle to be used when unregistering the alarm.

I/O Status Block

IoStatus.Status is set to STATUS_SUCCESS if the operation succeeded, or to an error status such as STATUS_INVALID_PARAMETER or STATUS_INSUFFICIENT_ RESOURCES otherwise.

IoStatus.Information is set to the length of the handle returned in **Irp->UserBuffer**.

See Also

SMB_REGISTER_ALARM, SMB_DEREGISTER_ALARM_NOTIFY

CHAPTER 2

SMB Structures

This chapter describes structures used by Windows 2000 SMB miniport clients, SMB miniport drivers, their support routines, and related IOCTLs. The following structures are described in alphabetical order:

SMB_CLASS
SMB_REGISTER_ALARM
SMB_REQUEST

These structures are defined in the header file *Smb.h*.

SMB_CLASS

```
typedef struct _SMB_CLASS {
  USHORT              MajorVersion;
  USHORT              MinorVersion;
  PVOID               Miniport;
  PDEVICE_OBJECT      DeviceObject;
  PDEVICE_OBJECT      PDO;
  PDEVICE_OBJECT      LowerDeviceObject;
  PIRP                CurrentIrp;
  PSMB_REQUEST        CurrentSmb;
  SMB_RESET_DEVICE    ResetDevice;
  SMB_START_IO        StartIo;
  SMB_STOP_DEVICE     StopDevice;
} SMB_CLASS, *PSMB_CLASS;
```

SMB_CLASS contains data shared by the SMB class driver and a miniport driver.

Members

MajorVersion

Major version number of the class driver. Must be SMB_CLASS_MAJOR_VERSION.

MinorVersion

Minor version number of the class driver. Must be SMB_CLASS_MINOR_VERSION.

Miniport

Pointer to extension data for the miniport driver.

DeviceObject

Pointer to the functional device object (FDO) for the miniport driver.

PDO

Pointer to the physical device object (PDO) for the miniport driver.

LowerDeviceObject

Pointer to the next lower device object in the device stack.

CurrentIrp

Pointer to the current IRP request, if any; otherwise, NULL.

CurrentSmb

Pointer to the current SMB_REQUEST in the current IRP request, if any; otherwise, NULL.

ResetDevice

Pointer to the miniport driver's SmbMiniResetDevice routine.

StartIo

Pointer to the miniport driver's SmbMiniStartIo routine.

StopDevice

Pointer to the miniport driver's SmbMiniStopDevice routine.

Comments

The class driver passes this structure in calls to the miniport driver's SmbMiniInitialize-Miniport routine.

The miniport driver passes this structure in calls to any of the class driver's routines.

SMB_REGISTER_ALARM

```
typedef struct {
  UCHAR               MinAddress;
  UCHAR               MaxAddress;
  SMB_ALARM_NOTIFY    NotifyFunction;
  PVOID               NotifyContext;
} SMB_REGISTER_ALARM, *PSMB_REGISTER_ALARM;
```

SMB_REGISTER_ALARM provides information required by the SMB class driver to register an alarm notification function.

Members

MinAddress

Specifies the lower limit of a range of bus addresses to which the notification function applies.

MaxAddress

Specifies the upper limit of a range of bus addresses to which the notification function applies.

NotifyFunction

Points to a notification function to be called when a miniport driver calls **SmbClassAlarm** to report an alarm. The function is declared as follows:

```
VOID
(*SMB_ALARM_NOTIFY) (
    PVOID       Context,
    UCHAR       Address,
    USHORT      Data
    );
```

Context

Context information passed through from the miniport driver.

Address

Bus address of the device that signaled the alarm.

Data

Alarm data passed through from the miniport driver.

NotifyContext

Points to the context data to be passed to the notification function.

Comments

An SMB miniport driver or the client of a miniport driver passes this structure when it requests alarm notification through the SMB_REGISTER_ALARM_NOTIFY IOCTL.

The values in **MinAddress** and **MaxAddress** define a range of bus addresses. If a device within the range signals an alarm, the SMB class driver calls the notification function.

See Also

SMB_REGISTER_ALARM_NOTIFY

SMB_REQUEST

```
typedef struct {
  UCHAR    Status;
  UCHAR    Protocol;
  UCHAR    Address;
  UCHAR    Command;
  UCHAR    BlockLength;
  UCHAR    Data[SMB_MAX_DATA_SIZE];
} SMB_REQUEST, *PSMB_REQUEST;
```

SMB_REQUEST provides information required to perform a specific I/O request on an SMB device.

Members

Status

Completion status of the SMB request.

Protocol

Specifies the bus protocol that applies to the current request. Possible values are:

SMB_WRITE_QUICK
SMB_READ_QUICK
SMB_SEND_BYTE
SMB_RECEIVE_BYTE
SMB_WRITE_BYTE
SMB_READ_BYTE
SMB_WRITE_WORD
SMB_READ_WORD
SMB_WRITE_BLOCK
SMB_READ_BLOCK
SMB_PROCESS_CALL

Address

Bus address of the device to which this request applies.

Command

Device-specific command to perform.

BlockLength

Number of bytes of data to which this request applies. This value is input for a write request and output for a read request.

Data

Array of data input or returned by this request.

Comments

Each client of an SMB miniport driver should define command codes that apply to its device.

See Also

SMB_BUS_REQUEST

PART 8

WMI Kernel-Mode Data Providers

CHAPTER 1

WMI IRPs

This chapter 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 that can be used by a DispatchSystemControl routine to handle WMI requests. If a driver registers as a WMI data provider by calling **IoWMIRegistrationControl**, it must handle such requests in one of the following ways:

- Call the kernel-mode WMI library routines declared in the *wmilib.h* header file. Drivers can use these routines only if they base static instance names on a single base name string or the device instance ID of a PDO. Drivers that use dynamic instance names can not use the WMI library routines.
- Process and complete any request that was tagged with a pointer to the driver's device object. Such a request is passed by the driver in its call to **IoWMIRegistrationControl**. Other IRP_MJ_SYSTEM_CONTROL requests must be forwarded to the next-lower driver.

The WMI library routines simplify the handling of WMI requests. Instead of processing each WMI request, a driver calls **WmiSystemControl** with a pointer to its device object, the IRP, and a WMILIB_CONTEXT structure. This WMILIB_CONTEXT structure contains pointers to a set of DpWmi*Xxx* callback routines that are defined by the driver. The WMI library validates the IRP parameters and calls the driver provided DpWmi*Xxx* routine for driver-specific processing. WMI library then packages any output in an appropriate WNODE_*XXX* structure. The output and status are returned to the caller. Drivers that use dynamic instance names must handle WMI requests by filling in the WNODE_*XXX* structure directly.

Drivers that do not register as WMI data providers must forward all WMI requests to the next-lower driver.

For information about registering as a WMI data provider, handling WMI IRPs, and using the WMI kernel-mode library routines, see the *Kernel-Mode Drivers Design Guide* in the online DDK.

IRP_MN_CHANGE_SINGLE_INSTANCE

All drivers that support WMI must handle this IRP.

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.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

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

A driver that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's DpWmiSetDataBlock routine, or returns STATUS_WMI_READ_ONLY to the caller if the driver does not define an entry point for such a routine.

A driver that handles an IRP_MN_CHANGE_SINGLE_INSTANCE request 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 forwards the request to the next-lower driver.

If the driver handles the request, it first checks the GUID at **Parameters.WMI.DataPath** to determine whether it identifies a data block supported by the driver. If not, the driver fails the IRP and returns STATUS_WMI_GUID_NOT_FOUND.

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 NUL terminator 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 a STATUS_WMI_INSTANCE_NOT_FOUND. In the case of an instance that has a dynamic instance name, this status indicates that the driver does not "own" 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 read/write data items in the instance to the values in the WNODE_SINGLE_INSTANCE, 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.

See Also

DpWmiSetDataBlock, **IoWMIRegistrationControl**, WMILIB_CONTEXT, **WmiSystemControl**, WNODE_SINGLE_INSTANCE

IRP_MN_CHANGE_SINGLE_ITEM

All drivers that support WMI must handle this IRP.

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.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

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_INSTANCE_ID_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

A driver that handles WMI IRPS by calling WMI library support routines calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's DpWmiSetDataItem routine, or returns STATUS_WMI_READ_ONLY to the caller if the driver does not define an entry point for such a routine.

A driver should handles an IRP_MN_CHANGE_SINGLE_ITEM request only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling a request, the driver determines whether **Parameters.WMI.DataPath** points to a GUID that the driver supports. If it does not, the driver fails the IRP and returns STATUS_WMI_GUID_NOT_FOUND.

If the driver supports the data block, it checks 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. **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). This length includes the NULL terminator 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. In the case of an instance with a dynamic instance name, this status indicates that the driver does not "own" 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.

See Also

DpWmiSetDataItem, **IoWMIRegistrationControl**, WMILIB_CONTEXT, **WmiSystemControl**, WNODE_SINGLE_ITEM

IRP_MN_DISABLE_COLLECTION

Any WMI driver that registers one or more of its data blocks as expensive to collect must handle this IRP.

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.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

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.

A driver that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's DpWmiFunctionControl routine, or simply returns STATUS_SUCCESS to the caller if the driver does not define an entry point for such a routine.

A driver handles an IRP_MN_DISABLE_COLLECTION request only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling the request, the driver determines whether **Parameters.WMI.DataPath** points to a GUID that the driver supports. If not, the driver fails the IRP and returns 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.

See Also

DpWmiFunctionControl, **IoWMIRegistrationControl**, IRP_MN_ENABLE_COLLECTION, WMILIB_CONTEXT, WMIREGGUID, WMIGUIDREGINFO, **WmiSystemControl**

IRP_MN_DISABLE_EVENTS

Any WMI driver that registers one or more event blocks must handle this IRP.

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.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

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 that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's DpWmiFunctionControl routine, or simply returns STATUS_SUCCESS to the caller if the driver does not define an entry point for such a routine.

A driver handles an IRP_MN_DISABLE_EVENTS request only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling a request, the driver determines whether **Parameters.WMI.DataPath** points to a GUID the driver supports. If not, the driver fails the IRP and returns 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 the *Kernel-Mode Drivers Design Guide* in the online DDK.

See Also

DpWmiFunctionControl, **IoWMIRegistrationControl**, IRP_MN_ENABLE_EVENTS, WMILIB_CONTEXT, **WmiSystemControl**

IRP_MN_ENABLE_COLLECTION

Any WMI driver that registers one or more of its data blocks as expensive to collect must handle this IRP.

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.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

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 that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's DpWmiFunctionControl

routine, or simply returns STATUS_SUCCESS to the caller if the driver does not define an entry point for such a routine.

A driver handles an IRP_MN_ENABLE_COLLECTION request only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver forwards 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 fails 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.

See Also

DpWmiFunctionControl, **IoWMIRegistrationControl**, IRP_MN_DISABLE_COLLECTION, WMILIB_CONTEXT, WMIREGGUID, **WmiSystemControl**

IRP_MN_ENABLE_EVENTS

Any WMI driver that registers one or more event blocks must handle this IRP.

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.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

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 that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's DpWmiFunctionControl routine, or simply returns STATUS_SUCCESS to the caller if the driver does not define an entry point for such a routine.

A driver handles an IRP_MN_ENABLE_EVENTS request only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver forwards 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 fails the IRP and returns 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 the *Kernel-Mode Drivers Design Guide* in the online DDK.

See Also

DpWmiFunctionControl, **IoWMIRegistrationControl**, IRP_MN_DISABLE_EVENTS, WMILIB_CONTEXT, **WmiSystemControl**, WNODE_EVENT_ITEM, WNODE_HEADER

IRP_MN_EXECUTE_METHOD

All drivers that support methods within data blocks must handle this IRP.

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.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

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_ITEM_ID_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 that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's DpWmiExecuteMethod routine, or returns STATUS_INVALID_DEVICE_REQUEST to the caller if the driver does not define an entry point for such a routine.

A driver handles an IRP_MN_EXECUTE_METHOD request only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling the request, the driver determines whether **Parameters.WMI.DataPath** points to a GUID supported by the driver. If not, the driver fails the IRP and returns 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 NUL terminator 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. In the case of a driver with a dynamic instance name, this status indicates that the driver does not "own" 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_ITEM_ID_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 non-zero, 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.

See Also

DpWmiExecuteMethod, **IoWMIRegistrationControl**, WMILIB_CONTEXT, **WmiSystemControl**, WNODE_METHOD_ITEM

IRP_MN_QUERY_ALL_DATA

All drivers that support WMI must handle this IRP.

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.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

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 an array of offsets to dynamic instance names.
- 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 OFFSETINSTANCE-DATA-ANDLENGTH 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 block uses dynamic instance names, writes the instance names at the offsets specified in the array at **OffsetInstanceNameOffsets**, with each dynamic name string aligned to a USHORT 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 that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's DpWmiQueryDataBlock routine.

A driver handles an IRP_MN_QUERY_ALL_DATA request only if **Parameters.WMI.ProviderId** points to the same device object that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling the request, the driver determines whether **Parameters.WMI.DataPath** points to a GUID that the driver supports. If not, the driver fails the IRP and returns STATUS_WMI_GUID_NOT_FOUND.

If the driver supports the data block, it fills in a WNODE_ALL_DATA structure at **Parameters.WMI.Buffer** with data for all instances of that data block.

See Also

DpWmiQueryDataBlock, **IoWMIRegistrationControl**, **KeQuerySystemTime**, WMILIB_CONTEXT, **WmiSystemControl**, WNODE_ALL_DATA

IRP_MN_QUERY_SINGLE_INSTANCE

All drivers that support WMI must handle this IRP.

When Sent

WMI sends this IRP to query for a single instance of a given data block.

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.

WMI sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

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

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, including instance data.
- Sets **SizeDataBlock** to the size in bytes of the instance data.
- 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 number of bytes written to the buffer at **Parameters.WMI.Buffer**.

Operation

A driver that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device

object, and a pointer to the IRP. **WmiSystemControl** calls the driver's DpWmiQueryDataBlock routine.

A driver handles an IRP_MN_QUERY_SINGLE_INSTANCE request 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 forwards the request to the next-lower driver.

Before handling the request, the driver determines whether **Parameters.WMI.DataPath** points to a GUID that the driver supports. If not, the driver fails the IRP and returns STATUS_WMI_GUID_NOT_FOUND.

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 NULL terminator 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. In the case of an instance with a dynamic instance name, this status indicates that the driver does not "own" 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.

See Also

DpWmiQueryDataBlock, **IoWMIRegistrationControl**, WMILIB_CONTEXT, **WmiSystemControl**, WNODE_SINGLE_INSTANCE

IRP_MN_REGINFO

All drivers that support WMI must handle this IRP.

When Sent

WMI sends this IRP to query or update a driver's registration information after the driver has called **IoWMIRegistrationControl**.

WMI sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input

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

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 that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's DpWmiQueryReginfo routine.

A driver handles an IRP_MN_REGINFO request only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWMIRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling the request, the driver checks **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_GUID. 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.

See Also

DpWmiQueryRegInfo, **IoWMIRegistrationControl**, WMILIB_CONTEXT, WMIREGGUID, WMIREGINFO, **WmiSystemControl**

CHAPTER 2

WMI Library Support Routines

This chapter describes the WMI library support routines that a driver can call to handle WMI IRPs. For information about handling WMI IRPs, see the *Kernel-Mode Drivers Design Guide* in the online DDK.

For information about WMI library callback routines, see Chapter 3.

WmiCompleteRequest

```
NTSTATUS
  WmiCompleteRequest(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN NTSTATUS Status,
    IN ULONG BufferUsed,
    IN CCHAR PriorityBoost
    );
```

WmiCompleteRequest indicates that a driver has finished processing a WMI request in a DpWmi*Xxx* routine.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

Status

Specifies the status to return for the IRP.

BufferUsed

Specifies the number of bytes needed in the buffer passed to the driver's DpWmiXxx routine. If the buffer is too small, the driver sets *Status* to STATUS_BUFFER_TOO_SMALL and sets *BufferUsed* to the number of bytes needed for the data to be returned. If the buffer passed is large enough, the driver sets *BufferUsed* to the number of bytes actually used.

PriorityBoost

Specifies a system-defined constant by which to increment the runtime priority of the original thread that requested the operation. WMI calls **IoCompleteRequest** with *PriorityBoost* when it completes the IRP.

Return Value

WmiCompleteRequest returns the value that was passed to it in the *Status* parameter unless *Status* was set to STATUS_BUFFER_TOO_SMALL.. If the driver set *Status* equal to STATUS_BUFFER_TOO_SMALL, **WmiCompleteRequest** builds a WNODE_TOO_SMALL structure and returns STATUS_SUCCESS. The return value from **WmiCompleteRequest** should be returned by the driver in its DpWmi*Xxx* routine.

Comments

A driver calls **WmiCompleteRequest** from a DpWmiXxx routine after it finishes all other processing in that routine, or after the driver finishes all processing for a pending IRP. **WmiCompleteRequest** fills in a WNODE_XXX with any data returned by the driver and calls **IoCompleteRequest** to complete the IRP.

A driver should always return the return value from **WmiCompleteRequest** in its DpWmi*Xxx* routine.

A driver must not call **WmiCompleteRequest** from its DpWmiQueryRegInfo routine.

Callers of **WmiCompleteRequest** must be running at IRQL <= DISPATCH_LEVEL.

See Also

DpWmiExecuteMethod, *DpWmiFunctionControl*, *DpWmiQueryDataBlock*, *DpWmiQueryReginfo*, *DpWmiSetDataBlock*, *DpWmiSetDataItem*, **IoCompleteRequest**, **WmiSystemControl**

WmiFireEvent

```
NTSTATUS
  WmiFireEvent(
    IN PDEVICE_OBJECT DeviceObject,
    IN LPGUID Guid,
    IN ULONG InstanceIndex,
    IN ULONG EventDataSize,
    IN PVOID EventData
    );
```

WmiFireEvent sends an event to WMI for delivery to data consumers that have requested notification of the event.

Parameters

DeviceObject

Points to the driver's device object.

Guid

Points to the GUID that represents the event block.

InstanceIndex

If the event block has multiple instances, specifies the index of the instance.

EventDataSize

Specifies the number of bytes of data at *EventData*. If no data is generated for an event, *EventData* must be zero.

EventData

Points to a driver-allocated nonpaged buffer containing data generated by the driver for the event. If no data is generated for an event, *EventData* must be NULL. WMI frees the buffer without further intervention by the driver.

Return Value

WmiFireEvent propagates the status returned by **IoWmiWriteEvent**, or returns STATUS_INSUFFICIENT_RESOURCES if it could not allocate memory for the event.

Comments

A driver calls **WmiFireEvent** to send an event to WMI for delivery to all data consumers that have requested notification of the event. All parameters passed to **WmiFireEvent** must be allocated from nonpaged pool.

The driver sends an event only if it has been previously enabled by the driver's DpWmiFunctionControl routine, which WMI calls to process an IRP_MN_ENABLE_EVENT request.

The driver writes any data associated with the event to the buffer at *EventData*. WMI fills in a WNODE_SINGLE_INSTANCE structure with the data and calls **IoWmiWriteEvent** to deliver the event.

Callers of **WmiFireEvent** must be running at IRQL <= DISPATCH_LEVEL.

See Also

DpWmiFunctionControl, IRP_MN_ENABLE_EVENTS, **WmiSystemControl**

WmiSystemControl

```
NTSTATUS
  WmiSystemControl(
    IN PWMILIB_CONTEXT WmiLibInfo,
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    OUT PSYSCTL_IRP_DISPOSITION IrpDisposition
    );
```

WmiSystemControl is a dispatch routine for drivers that use WMI library support routines to handle WMI IRPs.

Parameters

WmiLibInfo

Points to a WMILIB_CONTEXT structure that contains registration information for a driver's data blocks and event blocks and defines entry points for the driver's WMI library callback routines.

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

IrpDisposition

After **WmiSystemControl** returns, *IrpDisposition* indicates how the IRP was handled:

IrpProcessed

The IRP was processed and possibly completed. If the driver's DpWmiXxx routine called by **WMISystemControl** did not complete the IRP, the driver must call **WmiCompleteRequest** to complete the IRP after **WmiSystemControl** returns.

IrpNotCompleted

The IRP was processed but not completed, either because WMI detected an error and set up the IRP with an appropriate error code, or processed an IRP_MN_REGINFO request. The driver must complete the IRP by calling **IoCompleteRequest**.

IrpNotWmi

The IRP is not a WMI request (that is, WMI does not recognize the IRP's minor code). If the driver handles IRP_MJ_SYSTEM_CONTROL requests with this IRP_MN_XXX, it should handle the IRP; otherwise the driver should forward the IRP to the next lower driver.

IrpForward

The IRP is targeted to another device object (that is, the device object pointer at **Parameters.WMI.ProviderId** in the IRP does not match the pointer passed by the driver in its call to **IoWMIRegistrationControl**). The driver must forward the IRP to the next lower driver.

Return Value

WmiSystemControl returns STATUS_SUCCESS or one of the following error codes:

STATUS_INVALID_DEVICE_REQUEST
STATUS_WMI_GUID_NOT_FOUND
STATUS_WMI_INSTANCE_NOT_FOUND

Comments

When a driver receives an IRP_MJ_SYSTEM_CONTROL request with a WMI IRP minor code, it calls **WmiSystemControl** with a pointer to the driver's WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. The WMILIB_CONTEXT structure contains registration information for the driver's data blocks and event blocks and defines entry points for its WMI library callback routines.

WmiSystemControl confirms that the IRP is a WMI request and determines whether the block specified by the request is valid for the driver. If so, it processes the IRP by calling the appropriate DpWmiXxx entry point in the driver's WMILIB_CONTEXT structure. WMI is running at IRQL PASSIVE_LEVEL when it calls the driver's DpWmiXxx routine.

Callers of **WmiSystemControl** must be running at IRQL PASSIVE_LEVEL.

A driver must be running at IRQL PASSIVE_LEVEL when it forwards an IRP_MJ_SYSTEM_CONTROL request to the next-lower driver.

See Also

DpWmiExecuteMethod, *DpWmiFunctionControl*, *DpWmiQueryDataBlock*, *DpWmiQueryReginfo*, *DpWmiSetDataBlock*, *DpWmiSetDataItem*, WMILIB_CONTEXT

CHAPTER 3

WMI Library Callback Routines

This describes required and optional routines that a driver must implement to handle WMI IRPs by calling WMI library support routines. A driver sets entry points to its DpWmi*Xxx* routines in the WMILIB_CONTEXT structure the driver passes to **WmiSystemControl**.

A driver's DpWmi*Xxx* routines can have any names chosen by the driver writer.

For information about WMI library support routines, see Chapter 2. For information about handling WMI IRPs, see the *Kernel-Mode Drivers Design Guide* in the online DDK.

DpWmiExecuteMethod

```
NTSTATUS
  DpWmiExecuteMethod(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN ULONG GuidIndex,
    IN ULONG InstanceIndex,
    IN ULONG MethodId,
    IN ULONG InBufferSize,
    IN ULONG OutBufferSize,
    IN OUT PUCHAR Buffer
    );
```

A driver's DpWmiExecuteMethod routine executes a method associated with a data block. This routine is optional.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

GuidIndex

Specifies the data block by its index into the list of GUIDs provided by the driver in the WMILIB_CONTEXT structure it passed to **WmiSystemControl**.

InstanceIndex

If the block specified by *GuidIndex* has multiple instances, *InstanceIndex* specifies the instance.

MethodId

Specifies the ID of the method to execute. The driver defines the method ID as an item in a data block.

InBufferSize

Indicates the size in bytes of the input data. If there is no input data, *InBufferSize* is zero.

OutBufferSize

Indicates the number of bytes available in the buffer for output data.

Buffer

Points to a buffer that holds the input data and receives the output data, if any, from the method. If the buffer is too small to receive all of the output, the driver returns STATUS_BUFFER_TOO_SMALL and calls **WmiCompleteRequest** with the size required.

Return Value

DpWmiExecuteMethod returns STATUS_SUCCESS or an appropriate error code such as the following:

STATUS_BUFFER_TOO_SMALL
STATUS_INVALID_DEVICE_REQUEST
STATUS_WMI_INSTANCE_NOT_FOUND
STATUS_WMI_ITEM_ID_NOT_FOUND

Comments

WMI calls a driver's DpWmiExecuteMethod routine after the driver calls **WmiSystemControl** in response to an IRP_MN_EXECUTE_METHOD request.

If a driver does not implement a DpWmiExecuteMethod routine, it must set **ExecuteWmiMethod** to NULL in the WMILIB_CONTEXT the driver passes to **WmiSystemControl**. In this case, WMI returns STATUS_INVALID_DEVICE_REQUEST to the caller in response to any IRP_MN_EXECUTE_METHOD request.

If the method generates output, the driver should check the size of the output buffer in *OutBufferSize* 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 possibly return STATUS_BUFFER_TOO_SMALL) before resetting the counters. This ensures that WMI can safely re-send the request with a larger buffer.

After executing the method and writing output, if any, to the buffer, the driver calls **WmiCompleteRequest** to complete the request.

This routine can be pageable.

See Also

IRP_MN_EXECUTE_METHOD, WMILIB_CONTEXT, **WmiCompleteRequest**, **WmiSystemControl**

DpWmiFunctionControl

```
NTSTATUS
  DpWmiFunctionControl(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN ULONG GuidIndex,
    IN WMIENABLEDISABLECONTROL Function,
    IN BOOLEAN Enable
    );
```

A driver's DpWmiFunctionControl routine enables or disables notification of events. It also enables or disables data collection for data blocks that the driver registered as expensive to collect. This routine is optional.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

GuidIndex

Specifies the block by its index into the list of GUIDs provided by the driver in the WMILIB_CONTEXT structure it passed to **WmiSystemControl**.

Function

Specifies WmiEventControl to enable or disable an event, or WmiDataBlockControl to enable or disable data collection for a block that was registered as expensive to collect

(that is, a block for which the driver set WMIREG_FLAG_EXPENSIVE in **Flags** of the WMIGUIDREGINFO structure used to register the block).

Enable

Specifies TRUE to enable the event or data collection, or FALSE to disable it.

Return Value

DpWmiFunctionControl returns STATUS_SUCCESS or an appropriate error status such as:

STATUS_WMI_GUID_NOT_FOUND
STATUS_INVALID_DEVICE_REQUEST

Comments

WMI calls a driver's DpWmiFunctionControl routine after the driver calls **WmiSystemControl** in response to one of the following requests:

IRP_MN_ENABLE_COLLECTION
IRP_MN_DISABLE_COLLECTION
IRP_MN_ENABLE_EVENTS
IRP_MN_DISABLE_EVENTS

If a driver does not implement a DpWmiFunctionControl routine, it must set **WmiFunctionControl** to NULL in the WMILIB_CONTEXT the driver passes to **WmiSystemControl**. WMI returns STATUS_SUCCESS to the caller.

It is unnecessary for the driver to check whether events or data collection are already enabled for a block because WMI sends a single enable request when the first data consumer enables the block, and sends a single disable request when the last data consumer disables the block. WMI will not call DpWmiFunctionControl to enable a block without an intervening call to disable it.

After enabling or disabling the event or data collection for the block, the driver calls **WmiCompleteRequest** to complete the request.

This routine can be pageable.

See Also

IRP_MN_ENABLE_COLLECTION, IRP_MN_DISABLE_COLLECTION, IRP_MN_ENABLE_EVENTS, IRP_MN_DISABLE_EVENTS, WMILIB_CONTEXT, **WmiSystemControl**

DpWmiQueryDataBlock

```
NTSTATUS
  DpWmiQueryDataBlock(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN ULONG GuidIndex,
    IN ULONG InstanceIndex,
    IN ULONG InstanceCount,
    IN OUT PULONG InstanceLengthArray,
    IN ULONG BufferAvail,
    OUT PUCHAR Buffer
    );
```

A driver's DpWmiQueryDataBlock routine returns either a single instance or all instances of a data block. This routine is required.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

GuidIndex

Specifies the data block by its index into the list of GUIDs provided by the driver in the WMILIB_CONTEXT structure it passed to **WmiSystemControl**.

InstanceIndex

If DpWmiQueryDataBlock is called in response to an IRP_MN_QUERY_SINGLE_INSTANCE request, *InstanceIndex* specifies the instance to be queried. If DpWmiQueryDataBlock is called in response to an IRP_MN_QUERY_ALL_DATA REQUEST, *InstanceIndex* is zero.

InstanceCount

If DpWmiQueryDataBlock is called in response to an IRP_MN_QUERY_SINGLE_INSTANCE request, *InstanceCount* is 1. If DpWmiQueryDataBlock is called in response to an IRP_MN_QUERY_ALL_DATA REQUEST, *InstanceCount* is the number of instances to be returned.

InstanceLengthArray

Points to an array of ULONGs that indicate the length of each instance to be returned. If the buffer at *Buffer* is too small to receive all of the data, the driver sets *InstanceLengthArray* to NULL.

BufferAvail

Specifies the maximum number of bytes available to receive data in the buffer at *Buffer*.

Buffer

Points to the buffer to receive instance data. If the buffer is large enough to receive all of the data, the driver writes instance data to the buffer with each instance aligned on an 8-byte boundary. If the buffer is too small to receive all of the data, the driver calls **WmiCompleteRequest** with *BufferUsed* set to the size required.

Return Value

DpWmiQueryDataBlock returns STATUS_SUCCESS or an error status such as the following:

STATUS_BUFFER_TOO_SMALL
STATUS_WMI_GUID_NOT_FOUND
STATUS_WMI_INSTANCE_NOT_FOUND

If the driver cannot complete the request immediately, it can return STATUS_PENDING.

Comments

WMI calls a driver's DpWmiQueryDataBlock routine after the driver calls **WmiSystemControl** in response to an IRP_MN_QUERY_DATA_BLOCK or IRP_MN_QUERY_ALL_DATA request.

After writing instance data to the buffer, the driver calls **WmiCompleteRequest** to complete the request.

This routine can be pageable.

See Also

IRP_MN_QUERY_ALL_DATA, IRP_MN_QUERY_SINGLE_INSTANCE, WMILIB_CONTEXT, **WmiCompleteRequest**, **WmiSystemControl**

DpWmiQueryReginfo

```
NTSTATUS
  DpWmiQueryReginfo(
    IN PDEVICE_OBJECT DeviceObject,
    OUT PULONG RegFlags,
    OUT PUNICODE_STRING InstanceName,
    OUT PUNICODE_STRING *RegistryPath,
    OUT PUNICODE_STRING MofResourceName,
    OUT PDEVICE_OBJECT *Pdo
    );
```

A driver's DpWmiQueryReginfo routine provides information about the data blocks and event blocks to be registered by a driver. This routine is required.

Parameters

DeviceObject

Points to the driver's device object.

RegFlags

Indicates common characteristics of all blocks being registered. Any flag set in *RegFlags* is applied to all blocks. A driver can supplement *RegFlags* for a given block by setting **Flags** in the block's WMIGUIDREGINFO structure. For example, a driver might clear WMIREG_FLAG_EXPENSIVE in *RegFlags*, but set it in **Flags** to register a given block as expensive to collect.

The driver sets one of the following flags in *RegFlags*:

WMIREG_FLAG_INSTANCE_BASENAME

Requests WMI to generate static instance names from a base name provided by the driver at the *InstanceName*. WMI generates instance names by appending a counter to the base name.

WMIREG_FLAG_INSTANCE_PDO

Requests WMI to generate static instance names from the device instance ID for the PDO. If the driver sets this flag, it must also set *Pdo* to the PDO passed to the driver's AddDevice routine. WMI generates instance names from the device instance path of the PDO. Using the device instance path as a base for static instance names is efficient because such names are guaranteed to be unique. WMI automatically supplies a "friendly" name for the instance as an item in a data block that can be queried by data consumers.

A driver might also set one or more of the following flags in *RegFlags*, but more typically would set them in **Flags** of a block's WMIGUIDREGINFO structure:

WMIREG_FLAG_EVENT_ONLY_GUID

The blocks can be enabled or disabled as events only, and cannot be queried or set. If this flag is clear, the blocks can also be queried or set.

WMIREG_FLAG_EXPENSIVE

Requests WMI to send an IRP_MN_ENABLE_COLLECTION request the first time a data consumer opens a data block and an IRP_MN_DISABLE_COLLECTION request when the last data consumer closes the data block. This is recommended if collecting such data affects performance, because a driver need not collect the data until a data consumer explicitly requests it by opening the block.

WMIREG_FLAG_REMOVE_GUID

Requests WMI to remove support for the blocks. This flag is valid only in response to a request to update registration information (IRP_MN_REGINFO with **DataPath** set to WMIUPDATE).

InstanceName

Points to a single counted Unicode string that serves as the base name for all instances of all blocks to be registered by the driver. WMI frees the string with **ExFreePool**. If WMIREG_FLAG_INSTANCE_BASENAME is clear, *InstanceName* is ignored.

RegistryPath

Points to a counted Unicode string that specifies the registry path passed to the driver's **DriverEntry** routine.

MofResourceName

Points to a single counted Unicode string that indicates the name of the MOF resource attached to the driver's binary image file. Typically this string would be a static defined by the driver. WMI makes a copy of this string after the driver returns from this routine. This string can be dynamically allocated by the driver. In the case of an allocated string, the driver is responsible for freeing the string which should be done after WmiSystemControl returns. If the driver does not have a MOF resource attached, it can leave *MofResourceName* unchanged.

Pdo

Points to the physical device object (PDO) passed to the driver's AddDevice routine. If WMIREG_FLAG_INSTANCE_PDO is set, WMI uses the device instance path of this PDO as a base from which to generate static instance names. If WMIREG_FLAG_INSTANCE_PDO is clear, WMI ignores *Pdo*.

Return Value

DpWmiQueryReginfo always returns STATUS_SUCCESS

Comments

WMI calls a driver's DpWmiQueryReginfo after the driver calls **WmiSystemControl** in response to an IRP_MN_REGINFO request. WMI sends this IRP after a driver calls **IoWMIRegistrationControl** with WMIREG_ACTION_REGISTER, WMIREG_ACTION_REREGISTER, or WMIREG_ACTION_UPDATE.

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.

The driver provides new or updated registration information about individual blocks, or indicates blocks to remove, in the WMILIB_CONTEXT structure it passes to **WmiSystemControl**. After the initial call, which establishes the driver's registry path and MOF resource name, a driver's DpWmiQueryReginfo routine can change flags common to all of a driver's blocks, provide a different base name string used to generate instance names, or change the basis for instance names from a string to the device instance path of the PDO.

The driver must not return STATUS_PENDING or block the request. The driver must not complete the request by calling **WmiCompleteRequest** from its DpWmiQueryReginfo routine or by calling **IoCompleteRequest** after **WmiSystemControl** returns.

This routine can be pageable.

See Also

IoWMIRegistrationControl, IRP_MN_REGINFO, WMILIB_CONTEXT, WMIGUIDREGINFO, **WmiSystemControl**

DpWmiSetDataBlock

```
NTSTATUS
  DpWmiSetDataBlock(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN ULONG GuidIndex,
    IN ULONG InstanceIndex,
    IN ULONG BufferSize,
    IN PUCHAR Buffer
    );
```

A driver's DpWmiSetDataBlock routine changes all data items in a single instance of a data block. This routine is optional.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

GuidIndex

Specifies the data block by its index into the list of GUIDs provided by the driver in the WMILIB_CONTEXT structure it passed to **WmiSystemControl**.

InstanceIndex

If the block specified by *GuidIndex* has multiple instances, *InstanceIndex* specifies the instance.

BufferSize

Specifies the size in bytes of the buffer at *Buffer*.

Buffer

Points to a buffer that contains new values for the instance.

Return Value

DpWmiSetDataBlock returns STATUS_SUCCESS or 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

If the driver cannot complete the request immediately, it can return STATUS_PENDING.

Comments

WMI calls a driver's DpWmiSetDataItem routine after the driver calls **WmiSystemControl** in response to an IRP_MN_CHANGE_SINGLE_INSTANCE request. If a driver does not implement a DpWmiSetDataItem routine, it must set **SetWmiDataBlock** to NULL in the WMILIB_CONTEXT the driver passes to **WmiSystemControl**. WMI returns STATUS_READ_ONLY to the caller.

This routine can be pageable.

See Also

IRP_MN_CHANGE_SINGLE_INSTANCE, WMILIB_CONTEXT, **WmiSystemControl**

DpWmiSetDataItem

```
NTSTATUS
  DpWmiSetDataItem(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN ULONG GuidIndex,
    IN ULONG InstanceIndex,
    IN ULONG DataItemId,
    IN ULONG BufferSize,
    IN PUCHAR Buffer
    );
```

A driver's DpWmiSetDataItem changes a single data item in an instance of a data block. This routine is optional.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

GuidIndex

Specifies the data block by its index into the list of GUIDs provided by the driver in the WMILIB_CONTEXT structure it passed to **WmiSystemControl**.

InstanceIndex

If the block specified by *GuidIndex* has multiple instances, *InstanceIndex* specifies the instance.

DataItemId

Specifies the ID of the data item to set.

BufferSize

Specifies the size in bytes of the buffer at *Buffer*.

Buffer

Points to a buffer that contains the new value for the data item.

Return Value

DpWmiSetDataItem returns STATUS_SUCCESS or an appropriate error code such as the following:

STATUS_WMI_INSTANCE_NOT_FOUND
STATUS_WMI_INSTANCE_ID_NOT_FOUND
STATUS_WMI_GUID_NOT_FOUND
STATUS_WMI_READ_ONLY
STATUS_WMI_SET_FAILURE

Comments

WMI calls a driver's DpWmiSetDataItem routine after the driver calls **WmiSystemControl** in response to an IRP_MN_CHANGE_SINGLE_ITEM request.

If a driver does not implement a DpWmiSetDataItem routine, it must set **SetWmiDataItem** to NULL in the WMILIB_CONTEXT the driver passes to **WmiSystemControl**. WMI returns STATUS_READ_ONLY to the caller.

This routine can be pageable.

See Also

IRP_MN_CHANGE_SINGLE_ITEM, WMILIB_CONTEXT, **WmiSystemControl**

CHAPTER 4

WMI Structures

This describes, in alphabetic order, the structures that are used to pass WMI information between WMI and drivers that are kernel-mode data providers.

WMILIB_CONTEXT

```
typedef struct _WMILIB_CONTEXT {
    ULONG GuidCount;
    PWMIGUIDREGINFO GuidList;
    PWMI_QUERY_REGINFO QueryWmiRegInfo;
    PWMI_QUERY_DATABLOCK QueryWmiDataBlock;
    PWMI_SET_DATABLOCK SetWmiDataBlock;
    PWMI_SET_DATAITEM SetWmiDataItem;
    PWMI_EXECUTE_METHOD ExecuteWmiMethod;
    PWMI_FUNCTION_CONTROL WmiFunctionControl;
} WMILIB_CONTEXT, *PWMILIB_CONTEXT;
```

A WMILIB_CONTEXT structure provides registration information for a driver's data blocks and event blocks and defines entry points for the driver's WMI library callback routines.

Members

GuidCount

Specifies the number of blocks registered by the driver.

GuidList

Points to an array of **GuidCount** WMIGUIDREGINFO structures that contain registration information for each block.

QueryWmiRegInfo

Points to the driver's DpWmiQueryReginfo routine, which is a required entry point for drivers that call WMI library support routines.

QueryWmiDataBlock

Points to the driver's DpWmiQueryDataBlock routine, which is a required entry point for drivers that call WMI library support routines.

SetWmiDataBlock

Points to the driver's DpWmiSetDataBlock routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to NULL. In this case, WMI returns STATUS_WMI_READ_ONLY to the caller in response to any IRP_MN_CHANGE_SINGLE_INSTANCE request.

SetWmiDataItem

Points to the driver's DpWmiSetDataItem routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to NULL. In this case, WMI returns STATUS_WMI_READ_ONLY to the caller in response to any IRP_MN_CHANGE_SINGLE_ITEM request.

ExecuteWmiMethod

Points to the driver's DpWmiExecuteMethod routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to NULL. In this case, WMI returns STATUS_INVALID_DEVICE_REQUEST to the caller in response to any IRP_MN_EXECUTE_METHOD request.

WmiFunctionControl

Points to the driver's DpWmiFunctionControl routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to NULL. In this case, WMI returns STATUS_SUCCESS to the caller in response to any IRP_MN_ENABLE_*XXX* or IRP_MN_DISABLE_*XXX* request.

Comments

A driver that handles WMI IRPs by calling WMI library support routines stores an initialized WMILIB_CONTEXT structure (or a pointer to such a structure) in its device extension. A driver can use the same WMILIB_CONTEXT structure for multiple device objects if each device object supplies the same set of data blocks.

When the driver receives an IRP_MJ_SYSTEM_CONTROL request, it calls **WmiSystemControl** with a pointer to its WMILIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** determines whether the IRP contains a WMI request and, if so, handles the request by calling the driver's appropriate DpWmiXxx routine.

Memory for this structure can be allocated from paged pool.

See Also

DpWmiExecuteMethod, *DpWmiFunctionControl*, *DpWmiQueryReginfo*, *DpWmiQueryDataBlock*, *DpWmiSetDataBlock*, *DpWmiSetDataItem*, WMIGUIDREGINFO, **WmiSystemControl**

WMIGUIDREGINFO

```
typedef struct {
    LPCGUID Guid;
    ULONG InstanceCount;
    ULONG Flags;
} WMIGUIDREGINFO, *PWMIGUIDREGINFO;
```

A WMIGUIDREGINFO structure contains registration information for a given data block or event block exposed by a driver that uses the WMI library support routines.

Members

Guid

Points to the GUID that identifies the block. The memory that contains the GUID can be paged unless it is also used to call **WmiFireEvent**.

InstanceCount

Specifies the number of instances defined for the block.

Flags

Indicates characteristics of the block. WMI ORs **Flags** with the flags set by the driver in the *RegFlags* parameter of its DpWmiQueryReginfo routine, which apply to all of the data blocks and event blocks registered by the driver. **Flags** therefore supplements the driver's default settings for a given block.

A driver might set the following flag in **Flags**:

WMIREG_FLAG_INSTANCE_PDO

Requests WMI to generate static instance names from the device instance ID for the PDO. If this flag is set, the *Pdo* parameter of the driver's DpWmiQueryReginfo routine points to the PDO passed to the driver's AddDevice routine. WMI generates instance names from the device instance path of the PDO. Using the device instance path as a base for static instance names is efficient because such names are guaranteed to be unique. WMI automatically supplies a "friendly" name for the instance as an item in a data block that can be queried by data consumers.

A driver might also set one or more of the following flags:

WMIREG_FLAG_EVENT_ONLY_GUID

The block can be enabled or disabled as an event only, and cannot be queried or set. If this flag is clear, the block can also be queried or set.

WMIREG_FLAG_EXPENSIVE

Requests WMI to send an IRP_MN_ENABLE_COLLECTION request the first time a data consumer opens the data block and an IRP_MN_DISABLE_COLLECTION request when the last data consumer closes the data block. This is recommended if collecting such data affects performance, because a driver need not collect the data until a data consumer explicitly requests it by opening the block.

WMIREG_FLAG_REMOVE_GUID

Requests WMI to remove support for this block. This flag is valid only in response to a request to update registration information (IRP_MN_REGINFO with **DataPath** set to WMIUPDATE).

Comments

A driver that handles WMI IRPs by calling WMI library support routines builds An array of WMIGUIDREGINFO structures, one for each data block and event block to be registered. The driver sets the **GuidList** member of its WMILIB_CONTEXT structure to point to the first WMIGUIDREGINFO in the series.

Memory for this structure can be allocated from paged pool.

See Also

DpWmiQueryReginfo, IRP_MN_DISABLE_COLLECTION, IRP_MN_ENABLE_COLLECTION, IRP_MN_REGINFO, **WmiFireEvent**, WMILIB_CONTEXT

WMIREGGUID

```
typedef struct {
    GUID Guid;
    ULONG Flags;
    ULONG InstanceCount;
    union {
        ULONG InstanceNameList;
        ULONG BaseNameOffset;
        ULONG_PTR Pdo;
        ULONG_PTR InstanceInfo;
    };
} WMIREGGUID, *PWMIREGGUID
```

A WMIREGGUID contains new or updated registration information for a data block or event block.

Members

Guid

Specifies the GUID that represents the block to register or update.

Flags

Indicates characteristics of the block to register or update.

If a block is being registered with static instance names, a driver sets one of the following flags:

WMIREG_FLAG_INSTANCE_LIST

Indicates that the driver provides static instance names for this block in a static list following the WMIREGINFO structure in the buffer at **IrpStack->Parameters.WMI. Buffer**. If this flag is set, **InstanceNameList** is the offset in bytes from the beginning of the WMIREGINFO structure that contains this WMIREGGUID to a contiguous series of **InstanceCount** counted UNICODE strings.

WMIREG_FLAG_INSTANCE_BASENAME

Requests WMI to generate static instance names from a base name provided by the driver following the WMIREGINFO structure in the buffer at **IrpStack->Parameters.WMI.-Buffer**. WMI generates instance names by appending a counter to the base name. If this flag is set, **BaseNameOffset** is the offset in bytes from the beginning of the WMIREGINFO structure that contains this WMIREGGUID to a single counted UNICODE string that serves as the base name.

WMIREG_FLAG_INSTANCE_PDO

Requests WMI to generate static instance names from the device instance ID for the PDO. If this flag is set, **InstanceInfo** points to the PDO passed to the driver's AddDevice routine. WMI generates instance names from the device instance path of the PDO. Using the device instance path as a base for static instance names is efficient because such names are guaranteed to be unique. WMI automatically supplies a "friendly" name for the instance as an item in a data block that can be queried by data consumers.

If a block is being registered with dynamic instance names, WMIREG_FLAG_ INSTANCE_LIST, WMIREG_FLAG_INSTANCE_BASENAME, and WMIREG_ FLAG_INSTANCE_PDO must be clear.

A driver might also set one or more of the following flags:

WMIREG_FLAG_EVENT_ONLY_GUID

The block can be enabled or disabled as an event only, and cannot be queried or set. If this flag is clear, the block can also be queried or set.

WMIREG_FLAG_EXPENSIVE

Requests WMI to send an IRP_MN_ENABLE_COLLECTION request the first time a data consumer opens the data block and an IRP_MN_DISABLE_COLLECTION request when the last data consumer closes the data block. This is recommended if collecting such data affects performance, because a driver need not collect the data until a data consumer explicitly requests it by opening the block.

WMIREG_FLAG_REMOVE_GUID

Requests WMI to remove support for this block. This flag is valid only in response to a request to update registration information (IRP_MN_REGINFO with **DataPath** set to WMIUPDATE).

WMIREG_FLAG_TRACED_GUID

The block can be written only to a log file and can be accessed only through user-mode routines declared in *evntrace.h*. Only NT kernel-mode data providers set this flag.

WMIREG_FLAG_TRACE_CONTROL_GUID

The GUID acts as the control GUID for enabling or disabling the trace GUIDs associated with it in the MOF file. This flag is valid only if WMIREG_FLAG_TRACED_GUID is also set. Only NT kernel-mode data providers set this flag.

InstanceCount

Specifies the number of static instance names to be defined for this block. If the block is being registered with dynamic instance names, WMI ignores **InstanceCount**.

InstanceNameList

Indicates the offset in bytes from the beginning of the WMIREGINFO structure that contains this WMIREGGUID to a contiguous series of **InstanceCount** counted Unicode strings. This member is valid only if WMIREG_FLAG_INSTANCE_LIST is set in **Flags**. If the block is being registered with dynamic instance names, WMI ignores **InstanceNameList**.

BaseNameOffset

Indicates the offset in bytes from the beginning of the WMIREGINFO structure that contains this WMIREGGUID to a single counted UNICODE string that serves as a base for

WMI to generate static instance names. This member is valid only if WMIREG_FLAG_INSTANCE_BASENAME is set in **Flags**. If the block is being registered with dynamic instance names, WMI ignores **BaseNameOffset**.

Pdo

Points to the physical device object (PDO) passed to the driver's AddDevice routine. WMI uses the device instance path of this PDO as a base from which to generate static instance names. This member is valid only if WMIREG_FLAG_INSTANCE_PDO is set in **Flags**. If the block is being registered with dynamic instance names, WMI ignores **Pdo**.

InstanceInfo

Reserved for use by WMI.

Comments

A driver builds one or more WMIREGGUID structures in response to an IRP_MN_REGINFO request to register or update its blocks. The driver passes an array of such structures at the **WmiRegGuid** member of a WMIREGINFO structure, which the driver writes to the buffer at **IrpStack->Parameters.WMI.Buffer**.

A driver can register or update a block with either static or dynamic instance names. Static instance names provide best performance; however, dynamic instance names are preferred for data blocks if the number of instances or instance names change frequently. For more information about instance names, see the *Kernel-Mode Drivers Design Guide* in the online DDK.

See Also

IRP_MN_REGINFO, WMIREGINFO

WMIREGINFO

```
typedef struct {
    ULONG BufferSize;
    ULONG NextWmiRegInfo;
    ULONG RegistryPath;
    ULONG MofResourceName;
    ULONG GuidCount;
    WMIREGGUIDW WmiRegGuid[];
} WMIREGINFO, *PWMIREGINFO;
```

A WMIREGINFO structure contains information provided by a driver to register or update its data blocks and event blocks.

Members

BufferSize

Indicates the total size of the WMI registration data associated with this WMIREGINFO structure, calculated as follows: (**sizeof**(WMIREGINFO) + (**GuidCount** * **sizeof**-(WMIREGGUID) + *additionaldata*). Additional data might include items such as the MOF resource name, registry path, and static instance names for blocks.

NextWmiRegInfo

If a driver handles WMI requests on behalf of another driver, as a class driver might on behalf of a miniclass driver, **NextWmiRegInfo** indicates the offset in bytes from the beginning of this WMIREGINFO to the next WMIREGINFO structure that contains WMI registration information for the other driver. Otherwise, **NextWmiRegInfo** is zero.

RegistryPath

Indicates the offset in bytes from the beginning of this structure to a counted Unicode string that specifies the registry path passed to the driver's **DriverEntry** routine. The string must be aligned on a USHORT boundary. This member should be set only in response to a WMI registration request (IRP_MN_REGINFO with **DataPath** set to WMIREGISTER).

MofResourceName

Indicates the offset in bytes from the beginning of this structure to a counted Unicode string that specifies the name of the MOF resource in the driver's image file. The string must be aligned on a USHORT boundary. This member should be set only in response to a WMI registration request (IRP_MN_REGINFO with **DataPath** set to WMIREGISTER).

GuidCount

Indicates the number of WMIREGGUID structures in the array at **WmiRegGuid**.

WmiRegGuid

Is an array of **GuidCount** WMIREGGUID structures.

Comments

In response to a registration request (IRP_MN_REGINFO with **DataPath** set to WMIREGISTER), a driver builds at least one WMIREGINFO structure and writes the WMIREGINFO structure to the buffer at **IrpStack->Parameters.WMI.Buffer**. The WMIREGINFO structure contains an array of WMIREGGUID structures, one for each data block or event block exposed by the driver.

If the driver handles WMI requests on behalf of another driver, it builds another WMIREGINFO containing an array of WMIREGGUID structures for each block exposed by the other driver, sets the **NextWmiRegInfo** member of the first WMIREGINFO to an

offset in bytes from the beginning of the first WMIREGINFO to the beginning of the next WMIREGINFO in the buffer, and writes both structures to the buffer. The driver indicates the total size of both WMIREGINFO structures and associated data when calls **IoCompleteRequest** to complete the IRP.

A driver can use the same WMIREGINFO structure(s) to remove or update blocks in response to an update request (IRP_MN_REGINFO with **DataPath** set to WMIUPDATE). If WMIREG_FLAG_REMOVE_GUID is set in the **Flags** member of a WMIREGGUID, WMI removes that block from the list of blocks previously registered by the driver. If WMIREG_FLAG_REMOVE_GUID is clear, WMI updates registration information for that block only if other WMIREGGUID members have changed—otherwise, WMI does not change to its registration information for that block.

See Also

IoCompleteRequest, IRP_MN_REGINFO, WMIREGGUID

WNODE_ALL_DATA

```
typedef struct tagWNODE_ALL_DATA {
    struct _WNODE_HEADER WnodeHeader;
    ULONG DataBlockOffset;
    ULONG InstanceCount;
    ULONG OffsetInstanceNameOffsets;
    union {
        ULONG FixedInstanceSize;
        OFFSETINSTANCEDATAANDLENGTH OffsetInstanceDataAndLength[];
    };
} WNODE_ALL_DATA, *PWNODE_ALL_DATA;
```

A WNODE_ALL_DATA structure contains data for all instances of a data block or event block.

Members

WnodeHeader

Is a WNODE_HEADER structure that contains information common to all WNODE_*XXX* structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the WNODE_*XXX* data being passed or returned.

DataBlockOffset

Indicates the offset in bytes from the beginning of the WNODE_ALL_DATA structure to the beginning of data for the first instance.

InstanceCount

Indicates the number of instances whose data follows the fixed members of the WNODE_ALL_DATA in the buffer at **IrpStack->Parameters.WMI.Buffer**.

OffsetInstanceNameOffsets

Indicates the offset in bytes from the beginning of the WNODE_ALL_DATA to an array of offsets to dynamic instance names. Each instance name must be aligned on a USHORT boundary. If all instances to be returned have static instance names, WMI ignores **OffsetInstanceNameOffsets**.

FixedInstanceSize

Indicates the size of each instance to be returned if all such instances are the same size. This member is valid only if the driver sets WNODE_FLAG_FIXED_INSTANCE_SIZE in **WnodeHeader.Flags**.

OffsetInstanceDataAndLength

If instances to be returned vary in size, **OffsetInstanceDataAndLength** is an array of **InstanceCount** OFFSETINSTANCEDATAANDLENGTH structures that specify the offset in bytes from the beginning of the WNODE_ALL_DATA to the beginning of each instance and its length. OFFSETINSTANCEDATAANDLENGTH is defined as follows:

```
typedef struct {
    ULONG OffsetInstanceData;
    ULONG LengthInstanceData;
} OFFSETINSTANCEDATAANDLENGTH, *POFFSETINSTANCEDATAANDLENGTH
```

OffsetInstanceData

Indicates the offset in bytes from the beginning of the WNODE_ALL_DATA to the instance data.

LengthInstanceData

Indicates the length in bytes of the instance data.

Each instance must be aligned on a USHORT boundary. The **OffsetInstanceDataAndLength** member is valid only if the driver clears WNODE_FLAG_FIXED_INSTANCE_SIZE in **WnodeHeader.Flags**.

Comments

A driver fills in a WNODE_ALL_DATA structure in response to an IRP_MN_QUERY_ALL_DATA request. A driver might also generate a WNODE_ALL_DATA as an event.

After filling in the fixed members of the structure, a driver writes instance data and dynamic instance names (if any) at **DataBlockOffset** and **OffsetInstanceNameOffsets**, respectively, in the buffer at **IrpStack->Parameters.WMI.Buffer**. If WNODE_FLAG_

FIXED_INSTANCE_SIZE is clear, the first offset follows the last element of the **OffsetInstanceDataAndLength** array, plus padding so the data begins on an 8-byte boundary.

Instance names must be USHORT aligned. Instance data must be QUADWORD aligned.

See Also

IRP_MN_QUERY_ALL_DATA, WNODE_EVENT_ITEM, WNODE_HEADER

WNODE_EVENT_ITEM

```
typedef struct tagWNODE_EVENT_ITEM {
    struct _WNODE_HEADER WnodeHeader;
    // Rest of WNODE data indicated by flags in WnodeHeader
} WNODE_EVENT_ITEM, *PWNODE_EVENT_ITEM;
```

A WNODE_EVENT_ITEM contains data generated by a driver for an event.

WnodeHeader

Is a WNODE_HEADER structure that contains information common to all WNODE_*XXX* structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the WNODE_*XXX* data being passed or returned.

Comments

A WNODE_EVENT_ITEM contains whatever data the driver determines is appropriate for an event, in a WNODE_*XXX* structure that is appropriate for that data.

A driver generates only events that it has previously enabled in response to an IRP_MN_ENABLE_EVENTS request. To generate an event, a driver calls **IoWMIWriteEvent** and passes a pointer to the WNODE_EVENT_ITEM. WMI queues the event for delivery to all data consumers registered for that event.

For best performance, events should be small in size. However, if the amount of data for an event exceeds the maximum size defined in the registry, a driver can pass a WNODE_EVENT_REFERENCE, which WMI uses to query for the related WNODE_EVENT_ITEM. For more information about defining and generating WMI events, see the *Kernel-mode Drivers Design Guide* in the online DDK.

See Also

IoWMIWriteEvent, IRP_MN_ENABLE_EVENTS, WNODE_ALL_DATA, WNODE_EVENT_REFERENCE, WNODE_HEADER, WNODE_SINGLE_INSTANCE, WNODE_SINGLE_ITEM

WNODE_EVENT_REFERENCE

```
typedef struct tagWNODE_EVENT_REFERENCE {
    struct _WNODE_HEADER WnodeHeader;
    GUID TargetGuid;
    ULONG TargetDataBlockSize;
    union
    {
       ULONG TargetInstanceIndex;
       WCHAR TargetInstanceName[];
    };
} WNODE_EVENT_REFERENCE, *PWNODE_EVENT_REFERENCE;
```

A WNODE_EVENT_REFERENCE contains information that WMI can use to query for an event that exceeds the event size limit set in the registry.

Members

WnodeHeader

Is a WNODE_HEADER structure that contains information common to all WNODE_*XXX* structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the WNODE_*XXX* data being passed or returned.

TargetGuid

Indicates the GUID that represents the event to query.

TargetDataBlockSize

Indicates the size of the event.

TargetInstanceIndex

Indicates the index into the driver's list of static instance names for the event. This member is valid only if the event block was registered with static instance names and WNODE_FLAGS_STATIC_INSTANCE_NAMES is set in **WnodeHeader.Flags**.

TargetInstanceName

Indicates the dynamic instance name of the event as a counted Unicode string. This member is valid only if WNODE_FLAGS_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags** and the event block was registered with dynamic instance names.

Comments

If the amount of data for an event exceeds the maximum size set in the registry, a driver can generate a WNODE_EVENT_REFERENCE that specifies a WNODE_EVENT_ITEM that

WMI can query to obtain the event. For more information about defining and generating WMI events, see the *Kernel-Mode Drivers Design* in the online DDK.

See Also

WNODE_EVENT_ITEM, WNODE_HEADER

WNODE_HEADER

```
typedef struct _WNODE_HEADER {
    ULONG BufferSize;
    UINT_PTR ProviderId;
    union {
        ULONG64 HistoricalContext;
        struct {
            ULONG Version;
            ULONG Linkage;
        };
    };
    union {
        HANDLE KernelHandle;
        LARGE_INTEGER TimeStamp;
    };
    GUID Guid;
    ULONG ClientContext;
    ULONG Flags;
} WNODE_HEADER, *PWNODE_HEADER;
```

A WNODE_HEADER is the first member of all other WNODE_*XXX* structures. It contains information common to all such structures.

Members

BufferSize

Specifies the size in bytes of the nonpaged buffer to receive any WNODE_*XXX* data to be returned, including this WNODE_HEADER, additional members of a WNODE_*XXX* structure of the type indicated by **Flags**, and any WMI- or driver-determined data that accompanies that structure.

ProviderId

Reserved for WMI.

HistoricalContext

Reserved for WMI.

Version

Reserved for WMI.

Linkage

Reserved for WMI.

TimeStamp

Indicates the system time a driver collected the WNODE_*XXX* data, in units of 100 nanoseconds since 1/1/1601. A driver can call **KeQuerySystemTime** to obtain this value. If the block is to be written to a log file (WNODE_FLAG_LOG_WNODE), an NT driver might also set WNODE_FLAG_USE_TIMESTAMP in **Flags** to request the system logger to leave the value of **TimeStamp** unchanged.

KernelHandle

Reserved for WMI.

Guid

Indicates the GUID that represents the data block associated with the WNODE_*XXX* to be returned.

ClientContext

Reserved for WMI.

Flags

Indicates the type of WNODE_*XXX* structure that contains the WNODE_HEADER:

WNODE_FLAG_ALL_DATA

The rest of a WNODE_ALL_DATA structure follows the WNODE_HEADER in the buffer.

WMI sets this flag in the WNODE_HEADER it passes with an IRP_MN_QUERY_ALL_DATA request.

A driver sets this flag in the WNODE_HEADER of an event that consists of all instances of a data block. If the data block size is identical for all instances, a driver also sets WNODE_FLAG_FIXED_INSTANCE_SIZE.

WNODE_FLAG_EVENT_ITEM

A driver sets this flag to indicate that the WNODE_XXX structure was generated as an event. This flag is valid only if WNODE_FLAG_ALL_DATA, WNODE_FLAG_SINGLE_INSTANCE, or WNODE_FLAG_SINGLE_ITEM is also set.

WNODE_FLAG_EVENT_REFERENCE

The rest of a WNODE_EVENT_REFERENCE structure follows the WNODE_HEADER in the buffer.

A driver sets this flag when it generates an event that is larger than the maximum size specified in the registry for an event. WMI uses the information in the WNODE_EVENT_REFERENCE to request the event data and schedules such a request according to the value of WNODE_FLAG_SEVERITY_MASK.

WNODE_FLAG_METHOD_ITEM

The rest of a WNODE_METHOD_ITEM structure follows the WNODE_HEADER in the buffer.

WMI sets this flag in the WNODE_HEADER it passes with an IRP_MN_EXECUTE_METHOD request.

WNODE_FLAG_SINGLE_INSTANCE

The rest of a WNODE_SINGLE_INSTANCE structure follows the WNODE_HEADER in the buffer.

WMI sets this flag in the WNODE_HEADER it passes with a request to query or change an instance.

A driver sets this flag in the WNODE_HEADER of an event that consists of a single instance of a data block.

WNODE_FLAG_SINGLE_ITEM

The rest of a WNODE_SINGLE_ITEM structure follows the WNODE_HEADER in the buffer.

WMI sets this flag in the WNODE_HEADER it passes with a request to change an item.

A driver sets this flag in the WNODE_HEADER of an event that consists of a single data item.

WNODE_FLAG_TOO_SMALL

The rest of a WNODE_TOO_SMALL structure follows the WNODE_HEADER in the buffer.

A driver sets this flag when it passes a WNODE_TOO_SMALL, indicating that the buffer is too small for all of the WNODE_*XXX* data to be returned.

In addition, **Flags** might be set with one or more of the following flags that provide additional information about the WNODE_*XXX*:

WNODE_FLAG_FIXED_INSTANCE_SIZE

All instances of a data block are the same size. This flag is valid only if WNODE_FLAG_ALL_DATA is also set.

WNODE_FLAG_INSTANCES_SAME

The number of instances and the dynamic instance names in a WNODE_ALL_DATA to be returned are identical to those returned from the previous WNODE_ALL_DATA query. This flag is valid only if WNODE_FLAG_ALL_DATA is also set. This flag is ignored for data blocks registered with static instance names.

For optimized performance, a driver should set this flag if it can track changes to the number or names of its data blocks. WMI can then skip the processing required to detect and update dynamic instance names.

WNODE_FLAG_STATIC_INSTANCE_NAMES

The WNODE_*XXX* data to be returned does not include instance names.

WMI sets this flag before requesting WNODE_*XXX* data for data blocks registered with static instance names. After receiving the returned WNODE_*XXX* from the driver, WMI fills in the static instance names specified at registration before passing the returned WNODE_*XXX* to a data consumer.

WNODE_FLAG_PDO_INSTANCE_NAMES

Static instance names are based on the device instance ID of the PDO for the device. A driver requests such names by setting WMIREG_FLAG_INSTANCE_PDO in the WMIREGGUID it uses to register the block.

WMI sets this flag before requesting WNODE_*XXX* data for data blocks registered with PDO-based instance names.

WNODE_FLAG_SEVERITY_MASK

The driver-determined severity level of the event associated with a returned WNODE_EVENT_REFERENCE, with 0x00 indicating the least severe and 0xff indicating the most severe level.

WMI uses the value of this flag to prioritize its requests for the event data.

WNODE_FLAG_USE_TIMESTAMP

The system logger should not modify the value of **TimeStamp** set by the driver.

An NT driver might also set **Flags** to one or more of the following values for event blocks to be written to a system log file:

WNODE_FLAG_LOG_WNODE

An event block is to be sent to the system logger. The event header is a standard WNODE_HEADER structure. If the driver clears WNODE_FLAG_TRACED_GUID, the block will

also be sent to WMI for delivery to any data consumers that have enabled the event. The driver must allocate the WNODE_*XXX* from pool memory. WMI frees the memory after delivering the event to data consumers.

WNODE_FLAG_TRACED_GUID

An event block is to be sent only to the system logger. It does not get sent to WMI data consumers. The event header is an EVENT_TRACE_HEADER structure, declared in *evntrace.h*, instead of a WNODE_HEADER. The driver must allocate memory for the WNODE_*XXX* and free it after **IoWMIWriteEvent** returns. The driver can allocate such memory either from the stack or, to minimize the overhead of allocating and freeing the memory, from the driver's thread local storage if the driver creates and maintains its own thread pool.

WNODE_FLAG_USE_GUID_PTR

The **Guid** member points to a GUID in memory, rather than containing the GUID itself. The system logger dereferences the pointer before passing the data to the consumer. This flag is valid only if WNODE_FLAG_LOG_WNODE or WNODE_FLAG_TRACED_GUID are also set.

WNODE_FLAG_USE_MOF_PTR

Data that follows the fixed members of a WNODE_*XXX* structure consists of an array of MOF_FIELD structures, defined in *evntrace.h*, that contain pointers to data and sizes rather than the data itself. The array can contain up to MAX_MOF_FIELD elements. The system logger dereferences the pointers before passing the data to the consumer This flag is valid only for blocks registered with WMIREG_FLAG_TRACED_GUID.

Comments

In an IRP_MN_CHANGE_*XXX* or IRP_MN_EXECUTE_METHOD request, **BufferSize** in the IRP indicates the maximum size in bytes of the output buffer, while **BufferSize** in the input WNODE_HEADER for such a request indicates the size in bytes of the input data in the buffer.

See Also

IoWMIWriteEvent, **KeQuerySystemTime**, WNODE_ALL_DATA, WNODE_EVENT_ITEM, WNODE_EVENT_REFERENCE, WNODE_METHOD_ITEM, WNODE_SINGLE_INSTANCE, WNODE_SINGLE_ITEM, WNODE_TOO_SMALL

WNODE_METHOD_ITEM

```
typedef struct tagWNODE_METHOD_ITEM {
    struct _WNODE_HEADER WnodeHeader;
    ULONG OffsetInstanceName;
    ULONG InstanceIndex;
    ULONG MethodId;
    ULONG DataBlockOffset;
    ULONG SizeDataBlock;
    UCHAR VariableData[];
} WNODE_METHOD_ITEM, *PWNODE_METHOD_ITEM;
```

A WNODE_METHOD_ITEM indicates a method associated with an instance of a data block and contains any input data for the method.

Members

WnodeHeader

Is a WNODE_HEADER structure that contains information common to all WNODE_*XXX* structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the WNODE_*XXX* data being passed or returned.

OffsetInstanceName

Indicates the offset in bytes from the beginning of this structure to the dynamic instance name of this instance, aligned on a USHORT boundary. This member is valid only if WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags**. If the data block was registered with static instance names, WMI ignores **OffsetInstanceName**.

InstanceIndex

Indicates the index of this instance into the driver's list of static instance names for this data block. This member is valid only if the data block was registered with static instance names and WNODE_FLAG_STATIC_INSTANCE_NAME is set in **WnodeHeader.Flags**. If the data block was registered with dynamic instance names, WMI ignores **InstanceIndex**.

MethodId

Specifies the ID of the method to execute.

DataBlockOffset

Indicates the offset from the beginning of an input WNODE_METHOD_ITEM to input data for the method, or the offset from the beginning of an output WNODE_METHOD_ ITEM to output data from the method.

SizeDataBlock

Indicates the size of the input data in an input WNODE_METHOD_ITEM, or zero if there is no input. In an output WNODE_METHOD_ITEM, **SizeDataBlock** indicates the size of the output data, or zero if there is no output.

VariableData

Contains additional data, including the dynamic instance name if any, and the input for or output from the method aligned on an 8-byte boundary.

Comments

WMI passes a WNODE_METHOD_ITEM with an IRP_MN_EXECUTE_METHOD request to specify a method to execute in an instance of a data block, plus any input data required by the method.

If a method generates output, a driver overwrites the input data with the output at **DataBlockOffset** in the buffer at **IrpStack->Parameters.WMI.Buffer**, and sets **SizeDataBlock** in the WNODE_METHOD_ITEM to specify the size of the output data.

See Also

WNODE_HEADER

WNODE_SINGLE_INSTANCE

```
typedef struct tagWNODE_SINGLE_INSTANCE {
    struct _WNODE_HEADER WnodeHeader;
    ULONG OffsetInstanceName;
    ULONG InstanceIndex;
    ULONG DataBlockOffset;
    ULONG SizeDataBlock;
    UCHAR VariableData[];
} WNODE_SINGLE_INSTANCE, *PWNODE_SINGLE_INSTANCE;
```

A WNODE_SINGLE_INSTANCE contains values for all data items in one instance of a data block.

Members

WnodeHeader

Is a WNODE_HEADER structure that contains information common to all WNODE_*XXX* structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the WNODE_*XXX* data being passed or returned.

OffsetInstanceName

Indicates the offset from the beginning of this structure to the dynamic instance name of this instance, aligned on a USHORT boundary. This member is valid only if WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags**. If the data block was registered with static instance names, WMI ignores **OffsetInstanceName**.

InstanceIndex

Indicates the index of an instance registered with static instance names. This member is valid only if WNODE_FLAG_STATIC_INSTANCE_NAME is set in **WnodeHeader.Flags**. If the data block was registered with dynamic instance names, WMI ignores **InstanceIndex**.

DataBlockOffset

Indicates the offset from the beginning of this structure to the beginning of the instance.

SizeDataBlock

Indicates the size of the data block for this instance.

VariableData

Contains additional data, including the dynamic instance name if any, padding so the instance begins on an 8-byte boundary, and the instance of the data block to be returned.

Comments

WMI passes a WNODE_SINGLE_INSTANCE with an IRP_MN_CHANGE_SINGLE_INSTANCE request to set read-write data items in an instance of a data block. A driver can ignore values passed for read-only data items in the instance.

A driver fills in a WNODE_SINGLE_INSTANCE in response to an IRP_MN_QUERY_SINGLE_INSTANCE request or to generate an event that consists of a single instance.

See Also

WNODE_EVENT_ITEM, WNODE_HEADER

WNODE_SINGLE_ITEM

```
typedef struct tagWNODE_SINGLE_ITEM {
    struct _WNODE_HEADER WnodeHeader;
    ULONG OffsetInstanceName;
    ULONG InstanceIndex;
    ULONG ItemId;
    ULONG DataBlockOffset;
```

```
    ULONG SizeDataItem;
    UCHAR VariableData[];
} WNODE_SINGLE_ITEM, *PWNODE_SINGLE_ITEM;
```

A WNODE_SINGLE_ITEM contains the value of a single data item in an instance of a data block.

Members

WnodeHeader

Is a WNODE_HEADER structure that contains information common to all WNODE_*XXX* structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the WNODE_*XXX* data being passed or returned.

OffsetInstanceName

Indicates the offset from the beginning of this structure to the dynamic instance name, if any, aligned on a USHORT boundary. This member is valid only if WNODE_FLAG_STATIC_INSTANCE_NAMES is clear in **WnodeHeader.Flags**. If the data block was registered with static instance names, WMI ignores **OffsetInstanceName**.

InstanceIndex

Indicates the index into the driver's list of static instance names of this instance. This member is valid only if the data block was registered with static instance names and WNODE_FLAG_STATIC_INSTANCE_NAME is set in **WnodeHeader.Flags**. If the data block was registered with dynamic instance names, WMI ignores **InstanceIndex**.

ItemId

Specifies the ID of the data item to set.

DataBlockOffset

Indicates the offset from the beginning of this structure to the new value for the data item.

SizeDataItem

Indicates the size of the data item.

VariableData

Contains additional data, including the dynamic instance name if any, padding so the data value begins on an 8-byte boundary, and the new value for the data item.

Comments

WMI passes a WNODE_SINGLE_ITEM with an IRP_MN_CHANGE_SINGLE_ITEM request to set the value of a data item in an instance of a data block.

A driver builds a WNODE_SINGLE_ITEM to generate an event that consists of a single data item.

See Also

WNODE_EVENT_ITEM, WNODE_HEADER

WNODE_TOO_SMALL

```
typedef struct tagWNODE_TOO_SMALL {
    struct _WNODE_HEADER WnodeHeader;
    ULONG SizeNeeded;
} WNODE_TOO_SMALL, *PWNODE_TOO_SMALL;
```

A WNODE_TOO_SMALL indicates the size of the buffer needed to receive output from a request.

Members

WnodeHeader

Is a WNODE_HEADER structure that contains information common to all WNODE_*XXX* structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the WNODE_*XXX* data being passed or returned.

SizeNeeded

Specifies the size of the buffer needed to receive all of the WNODE_*XXX* data to be returned.

Comments

When the buffer for a WMI request is too small to receive all of the data to be returned, a driver fills in a WNODE_TOO_SMALL structure to indicate the required buffer size. WMI can then increase the buffer to the recommended size and issue the request again. A driver is responsible for managing any side effects caused by handling the same request more than once.

See Also

WNODE_HEADER

CHAPTER 5

WMI Event Trace Structures

This section describes the structure that is used to send WMI events to the WMI event logger.

EVENT_TRACE_HEADER

```
typedef struct _EVENT_TRACE_HEADER {
    USHORT Size;
    UCHAR HeaderType;
    UCHAR MarkerFlags;
    union {
        ULONG Version;
        struct {
            UCHAR Type;
            UCHAR Level;
            USHORT Version;
        } Class;
    };
    ULONGLONG ThreadId;
    LARGE_INTEGER TimeStamp;
    union {
        GUID Guid;
        ULONGULONG GuidPtr;
    };
    union {
        struct {
            ULONG ClientContext;
            ULONG Flags;
        };
        struct {
            ULONG KernelTime;
            ULONG UserTime;
        };
    ULONG64 ProcessorTime;
} EVENT_TRACE_HEADER; *PEVENT_TRACE_HEADER;
```

An EVENT_TRACE_HEADER structure is used to pass a WMI event to the WMI event logger. It is overlaid on the WNODE_HEADER portion of the WNODE_EVENT_ITEM passed to **IoWMIFireEvent**. Information contained in the EVENT_TRACE_HEADER is written to the WMI log file.

Members

Size

Specifies the size in bytes of this structure. This value should be set to - SIZEOF(EVENT_TRACE_HEADER) plus the size of any driver data appended to the end of this structure. (Note: The size of this member is smaller than the size of the **Size** member of the WNODE_HEADER structure on which this structure is overlaid.)

HeaderType

Reserved for internal use.

MarkerFlags

Reserved for internal use.

Version

Drivers can use this member to store version information. This information is not interpreted by the event logger.

Class

Type

Trace event type. This can be one of the predefined EVENT_TRACE_TYPE_Xxx values contained in evntrace.h or can be a driver defined value. Callers are free to define private event types with values greater than the reserved values in evntrace.h.

Level

Trace instrumentation level. A driver defined value meant to represent the degree of detail of the trace instrumentation. Drivers are free to give this value meaning. This value should be zero by default. More information on how consumers can request different levels of trace information will be provided in a future version of the documentation.

Version

Version of trace record. Version information that can be used by the driver to track different event formats.

ThreadId

Reserved for internal use.

TimeStamp

Indicates the time the driver event occurred. This time is indicated in units of 100 nano-seconds since 1/1/1601. If the WNODE_FLAG_USE_TIMESTAMP is set in **Flags,** the system logger will leave the value of TimeStamp unchanged. Otherwise, the system logger will set the value of **TimeStamp** at the time it receives the event. A driver can call **KeQuerySystemTime** to set the value of **TimeStamp**.

Guid

Indicates the GUID that identifies the data block for the event.

GuidPtr

If the WNODE_FLAG_USE_GUID_PTR is set in **Flags, GuidPtr** points to the GUID that identifies the data block for the event.

ClientContext

Reserved for internal use.

Flags

Provides information about the structure's contents. For information on EVENT_TRACE_ HEADER **Flags** values, see the **Flags** description in **WNODE_HEADER** in Chapter 4.

KernelTime

Reserved for internal use.

UserTime

Reserved.

ProcessorTime

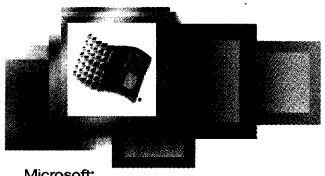
Reserved for internal use.

Comments

A driver which supports trace events will use this structure to report events to the WMI event logger. Trace events should not be reported until the driver receives a request to enable events and the control GUID is one the driver supports. The driver should initialize an EVENT_TRACE_HEADER structure, fill in any user defined event data at the end and pass a pointer to the EVENT_TRACE_HEADER to **IoWmiWriteEvent**.. The driver should continue reporting trace events until it receives a request to disable the control GUID for the trace events.

See Also

WNODE_HEADER, WNODE_EVENT_ITEM, IoWMIWriteEvent



Microsoft Windows 2000

Driver Development Reference Volume 2

The essential reference to kernel-mode drivers

Developing reliable drivers—the most essential part of any operating system—requires good documentation. Open this volume to get complete, authoritative reference information about kernel-mode drivers, including drivers for input devices, devices that use serial and parallel ports, and devices that use USB, IEEE 1394, and PCMCIA ports.

mspress.microsoft.com

Part No. 097-0002734

Microsoft®

Driver Development Reference Volume 2

The essential reference to kernel-mode drivers

Developing reliable drivers—the most essential part of any operating system—requires good documentation. Open this volume to get complete, authoritative reference information about kernel-mode drivers, including drivers for input devices, devices that use serial and parallel ports, and devices that use USB, IEEE 1394, and PCMCIA ports.

mspress.microsoft.com

Part No. 097-0002734

Microsoft®