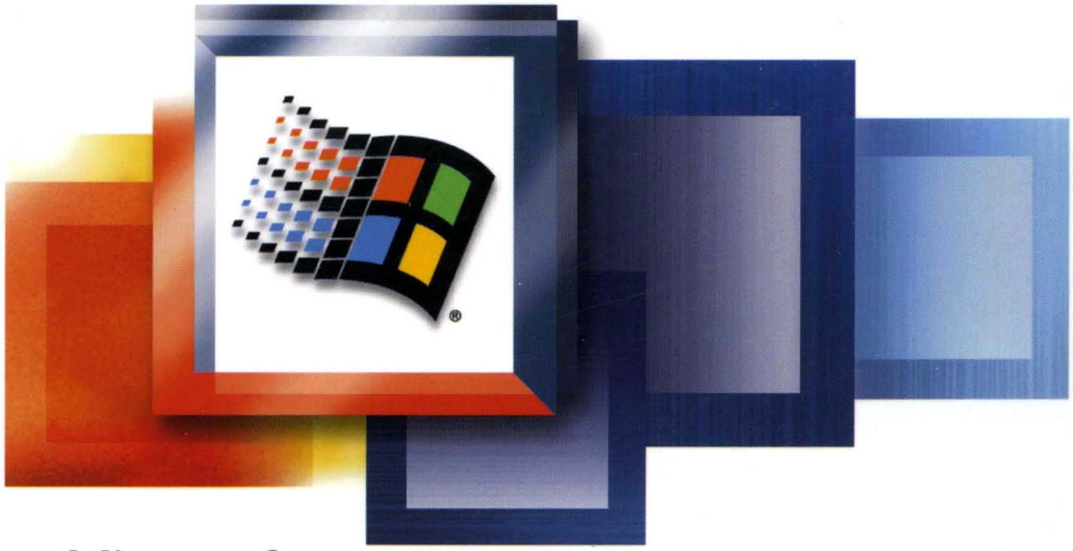
MICROSOFT® PROFESSIONAL EDITIONS

Microsoft®

CD-ROM
Included

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



Microsoft® Windows® 2000 Driver Development Reference

Volume 1

Microsoft®

Driver Development Reference

Volume 1

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.

1 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.

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. 097-0002733

Contents

P A R T 1

Plug and Play

C H A P T E R 1

Plug and Play Routines

These routines are used by drivers to implement plug and play support. The routines are listed in alphabetical order. The following lists summarize the routines functionally.

See the *Plug and Play, Power Management, and Setup Design Guide* for background and task-oriented information on supporting PnP in drivers.

Device Information Routines

IoGetDeviceProperty

Retrieves information about a device such as configuration information and the name of its PDO.

IoInvalidateDeviceRelations

Notifies the PnP Manager that the relations for a device have changed.

IoInvalidateDeviceState

Notifies the PnP Manager that 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.

IoReportDetectedDevice

Reports a non PnP 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.

Registry Routines

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.

Device Interface Routines

IoRegisterDeviceInterface

Registers device functionality (a device interface) that a driver will 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.

IoOpenDeviceInterfaceRegistryKey

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

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).

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.

PnP Notification Routines

IoRegisterPlugPlayNotification

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

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 it. Do not use this routine to report system PnP events, such as GUID_TARGET_DEVICE_REMOVE_COMPLETE.

IoReportTargetDeviceChangeAsynchronous

Notifies the PnP Manager that a custom event has occurred on a device. Returns immediately; does not wait while the PnP Manager sends notification of the event to drivers that registered for it. Do not use this routine to report system PnP events, such as GUID_TARGET_DEVICE_REMOVE_COMPLETE.

IoUnregisterPlugPlayNotification

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

Remove Lock Routines

IoInitializeRemoveLock

Initalizes a remove lock for a device object. A driver can use the lock to track outstanding I/O on a device and to determine when the driver can delete its device object in response to an IRP_MN_REMOVE_DEVICE request.

IoAcquireRemoveLock

Increments the count for a remove lock, indicating that the associated device object should not be detached from the device stack nor deleted.

IoReleaseRemoveLock

Releases a remove lock acquired with a previous call to **IoAcquireRemoveLock**.

IoReleaseRemoveLockAndWait

Releases a remove lock acquired with a previous call to **IoAcquireRemoveLock** and waits until all acquisitions of the lock have been released. A driver typically calls this routine in its dispatch code for an IRP_MN_REMOVE_DEVICE request.

Other PnP Routines

IoAdjustPagingPathCount

Increments or decrements a caller-supplied page-file counter as an atomic operation. This routine can be used to adjust other counters, such as counters for hibernation files or crash-dump files.

IoRequestDeviceEject

Notifies the PnP manager that the device eject button was pressed. Note that this routine reports a request for a device eject, not media eject.

IoAcquireRemoveLock

```
NTSTATUS
  IoAcquireRemoveLock(
    IN PIO_REMOVE_LOCK RemoveLock,
    IN OPTIONAL PVOID Tag
    );
```

IoAcquireRemoveLock increments the count for a remove lock, indicating that the associated device object should not be detached from the device stack nor deleted.

Parameters

RemoveLock

Points to an IO_REMOVE_LOCK structure that the caller initialized with a previous call to **IoInitializeRemoveLock**.

Tag

Optionally points to a caller-supplied tag that identifies this instance of acquiring the remove lock. For example, a driver Dispatch routine typically sets this parameter to a pointer to the IRP the routine is processing.

If a driver specifies a *Tag* on a call to **IoAcquireRemoveLock**, the driver must supply the same *Tag* in the corresponding call to **IoReleaseRemoveLock**.

The *Tag* does not have to be unique, but should be something meaningful during debugging.

The I/O system only uses this parameter on checked builds.

Include

ntddk.h

Return Value

IoAcquireRemoveLock returns STATUS_SUCCESS if the call was successful. Possible error return values include:

Error Status	Description
STATUS_DELETE_PENDING	The driver has received an IRP_MN_REMOVE_DEVICE for the device and is waiting for all remove locks to clear before deleting the device object. Do not start any new operations on the device.

Comments

A driver must initialize a remove lock with a call to **IoInitializeRemoveLock** before using the lock.

A driver must call **IoReleaseRemoveLock** to release the lock when it is no longer needed.

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

See Also

IoInitializeRemoveLock, **IoReleaseRemoveLock**, **IoReleaseRemoveLockAndWait**

IoAcquireRemoveLockEx

This routine is reserved for system use. See **IoAcquireRemoveLock**.

IoAdjustPagingPathCount

```
VOID
  IoAdjustPagingPathCount(
    IN PLONG Count,
    IN BOOLEAN Increment
    );
```

IoAdjustPagingPathCount increments or decrements a caller-supplied page-file counter as an atomic operation. This routine can be used to adjust other counters, such as counters for hibernation files or crash-dump files.

Parameters

Count

Points to a caller-supplied variable that contains a counter. A driver typically stores a page-file counter in the device extension for the device.

Increment

Specifies whether the counter is to be incremented or decremented. A value of TRUE specifies an increment operation.

Include

wdm.h or *ntddk.h*

Comments

This routine is useful for maintaining a count of paging files on a device. The operating system notifies a driver that a paging file has been created on, or removed from, one of the driver's devices by sending an IRP. The IRP has the major code IRP_MJ_PNP and the minor code IRP_MN_DEVICE_USAGE_NOTIFICATION.

This routine can be used for other counters, such as counters for hibernation files or crash-dump files.

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

See Also

IRP_MN_DEVICE_USAGE_NOTIFICATION

IoGetDeviceInterfaceAlias

```
NTSTATUS
  IoGetDeviceInterfaceAlias(
    IN PUNICODE_STRING SymbolicLinkName,
    IN CONST GUID *AliasInterfaceClassGuid,
    OUT PUNICODE_STRING AliasSymbolicLinkName
    );
```

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.

Parameters

SymbolicLinkName

Points to the name of the device interface for which to retrieve an alias. The caller typically received this string from a call to **IoGetDeviceInterfaces** or in a PnP notification structure.

AliasInterfaceClassGuid

Points to a GUID specifying the interface class of the alias to retrieve.

AliasSymbolicLinkName

Specifies a pointer to a NULL unicode string. On successful return, *AliasSymbolicLink-Name*.**Buffer** points to a string containing the name of the requested alias. The caller must free the unicode string with **RtlFreeUnicodeString** when it is no longer needed.

Include

wdm.h or *ntddk.h*

Return Value

IoGetDeviceInterfaceAlias returns STATUS_SUCCESS if the call was successful. Possible error return values include:

Error Status	**Description**
STATUS_OBJECT_NAME_NOT_FOUND	Possibly indicates that there is no alias of the specified interface class.
STATUS_OBJECT_PATH_NOT_FOUND	Possibly indicates that there is no alias of the specified interface class.
STATUS_INVALID_HANDLE	Possibly indicates an invalid *SymbolicLinkName* or an invalid *AliasClassGuid*.

Comments

The *SymbolicLinkName* parameter specifies a device interface for a particular device, belonging to a particular interface class, with a particular reference string. **IoGetDevice-InterfaceAlias** returns another device interface for the same device and reference string, but of a different interface class, if it exists.

For example, the function driver for a fault-tolerant volume could register and set two device interfaces, one of the fault-tolerant-volume interface class and one of the volume interface class. Another driver could call **IoGetDeviceInterfaceAlias** with the symbolic link for one of the interfaces and ask whether the other interface exists by specifying its interface class.

Two device interfaces with NULL reference strings are aliases if they are exposed by the same underlying device and have different interface class GUIDs.

Callers of **IoGetDeviceInterfaceAlias** must be running at IRQL PASSIVE_LEVEL in the context of a system thread.

See Also

IoRegisterDeviceInterface, **RtlFreeUnicodeString**

IoGetDeviceInterfaces

```
NTSTATUS
  IoGetDeviceInterfaces(
    IN CONST GUID *InterfaceClassGuid,
    IN PDEVICE_OBJECT PhysicalDeviceObject  OPTIONAL,
    IN ULONG Flags,
    OUT PWSTR *SymbolicLinkList
    );
```

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).

Parameters

InterfaceClassGuid

Points to a class GUID specifying the device interface class. The GUIDs for a class should be in a device-specific .h file.

PhysicalDeviceObject

Points to an optional PDO that narrows the search to only the device interfaces of the device represented by the PDO.

Flags

Specifies flags that modify the search for device interfaces.

Flag	Meaning
DEVICE_INTERFACE_INCLUDE_ NONACTIVE	Return disabled device interfaces in addition to enabled interfaces.

When searching for a device that supports a particular interface, the caller requires an enabled interface and thus does not set the DEVICE_INTERFACE_INCLUDE_ NONACTIVE flag.

A driver typically sets the DEVICE_INTERFACE_INCLUDE_NONACTIVE flag to locate disabled interfaces that the driver must enable. For example, the class installer for the device may have been directed by the INF file to register one or more interfaces for the device. The interfaces would be registered but are not usable until they are enabled by the driver (using **IoSetDeviceInterfaceState**). To narrow the list of interfaces returned to only those exposed by a given device, a driver can specify a *PhysicalDeviceObject*.

SymbolicLinkList

Points to a character pointer that is filled in on successful return with a list of unicode strings identifying the device interfaces that match the search criteria. The newly allocated buffer contains a list of symbolic link names. Each unicode string in the list is null-terminated; the end of the whole list is marked by an additional NULL. The caller is responsible for freeing the buffer (**ExFreePool**) when it is no longer needed.

If no device interfaces match the search criteria, this routine returns STATUS_SUCCESS and the string contains a single NULL character.

Include

wdm.h or *ntddk.h*

Return Value

IoGetDeviceInterfaces returns STATUS_SUCCESS if the call was successful. Possible error return values include:

Error Status	Description
STATUS_INVALID_DEVICE_REQUEST	Possibly indicates that *PhysicalDeviceObject* was not a valid PDO pointer.

Comments

IoGetDeviceInterfaces returns a list of device interfaces that match the search criteria. A kernel-mode component typically calls this routine to get a list of all enabled device interfaces of a particular device interface class. Such a component can get a pointer to the file object and/or the device object for an interface using **IoGetDeviceObjectPointer** or **ZwCreateFile**. The device object pointer returned by **IoGetDeviceObjectPointer** points to the top of the device stack for the device and can be used in calls to **IoCallDriver**.

If there is a default interface for the requested device interface class, it is listed first in *SymbolicLinkList*. Default interfaces can be set by user mode, but not by kernel mode.

The format of a symbolic link name is opaque; the caller should not attempt to parse a symbolic link name.

Symbolic links for device interfaces can be used across system boots.

To be notified when additional device interfaces of a particular class are enabled on the system, register for notification of a device class change with **IoRegisterPlugPlay-Notification**.

Callers of **IoGetDeviceInterfaces** must be running at IRQL PASSIVE_LEVEL in the context of a system thread.

See Also

ExFreePool, **IoGetDeviceObjectPointer**, **IoRegisterDeviceInterface**, **IoRegister-PlugPlayNotification**, **IoSetDeviceInterfaceState**, **ZwCreateFile**

IoGetDeviceProperty

```
NTSTATUS
  IoGetDeviceProperty(
    IN PDEVICE_OBJECT DeviceObject,
    IN DEVICE_REGISTRY_PROPERTY DeviceProperty,
    IN ULONG BufferLength,
    OUT PVOID PropertyBuffer,
    OUT PULONG ResultLength
    );
```

IoGetDeviceProperty retrieves information about a device such as configuration information and the name of its PDO.

Parameters

DeviceObject

Points to the physical device object (PDO) for the device being queried.

DeviceProperty

Specifies the device property being requested. Must be one of the following:

DevicePropertyAddress

Requests the address of the device on the bus. *PropertyBuffer* points to a ULONG.

The interpretation of this address is bus-specific. The caller of this routine should call the routine again to request the **DevicePropertyBusTypeGuid**, or possibly the **DevicePropertyLegacyBusType**, so it can interpret the address. An address value of 0xFFFFFFFF indicates that the underlying bus driver did not supply a bus address for the device.

DevicePropertyBootConfiguration

Requests the hardware resources assigned to the device by the firmware, in raw form. *PropertyBuffer* points to a CM_RESOURCE_LIST.

DevicePropertyBootConfigurationTranslated

The hardware resources assigned to the device by the firmware, in translated form. *PropertyBuffer* points to a CM_RESOURCE_LIST.

DevicePropertyBusNumber

Requests the legacy bus number of the bus the device is connected to. *PropertyBuffer* points to a ULONG.

DevicePropertyBusTypeGuid

Requests the GUID for the bus that the device is connected to. The system-defined bus type GUIDs are listed in *wdmguid.h*. *PropertyBuffer* points to a GUID, which is a 16-byte structure that contains the GUID in binary form.

DevicePropertyClassGuid

Requests the GUID for the device's setup class. *PropertyBuffer* points to a NUL-terminated array of WCHAR. This routine returns the GUID in a string format as follows, where each "c" represents a hexadecimal character: {cccccccc-cccc-cccc-cccc-cccccccccccc}

DevicePropertyClassName

Requests the name of the device's setup class, in text format. *PropertyBuffer* points to a NUL-terminated array of WCHAR.

DevicePropertyCompatibleIDs

Requests the compatible IDs reported by the device. *PropertyBuffer* points to a MULTI_SZ.

DevicePropertyDeviceDescription

Requests a string describing the device, such as "Microsoft PS/2 Port Mouse", typically defined by the manufacturer. *PropertyBuffer* points to a NUL-terminated array of WCHAR.

DevicePropertyDriverKeyName

Requests the name of the driver-specific registry key. *PropertyBuffer* points to a NUL-terminated array of WCHAR.

DevicePropertyEnumeratorName

Requests the name of the enumerator for the device, such as "PCI" or "root". *PropertyBuffer* points to NUL-terminated array of WCHAR.

DevicePropertyFriendlyName

Requests a string that can be used to distinguish between two similar devices, typically defined by the class installer. *PropertyBuffer* points to a NUL-terminated array of WCHAR.

DevicePropertyHardwareID

Requests the hardware IDs provided by the device that identify the device. *PropertyBuffer* points to a MULTI_SZ.

DevicePropertyLegacyBusType

Requests the bus type, such as PCIBus or PCMCIABus. *PropertyBuffer* points to an INTERFACE_TYPE.

DevicePropertyLocationInformation

Requests information about the device's location on the bus; the interpretation of this information is bus-specific. *PropertyBuffer* points to a NUL-terminated array of WCHAR.

DevicePropertyManufacturer

Requests a string identifying the manufacturer of the device. *PropertyBuffer* points to a NUL-terminated array of WCHAR.

DevicePropertyPhysicalDeviceObjectName

Requests the name of the PDO for this device. *PropertyBuffer* points to a NUL-terminated array of WCHAR.

DevicePropertyUINumber

Requests a number associated with the device that can be displayed in the user interface. *PropertyBuffer* points to a ULONG.

This number is typically a user-perceived slot number, such as a number printed next to the slot on the board, or some other number that makes locating the physical device easier for the user. If the device is on a bus that has no UI number convention, or if the bus driver for the device cannot determine the UI number, this value is 0xFFFFFFFF.

BufferLength

Specifies the size, in bytes, of the caller-supplied *PropertyBuffer*.

PropertyBuffer

Points to a caller-supplied buffer to receive the property information. The buffer can be allocated from pageable memory. The type of the buffer is determined by the *Device-Property* (see above).

ResultLength

Points to a ULONG to receive the size of the property information returned at *Property-Buffer*. If **IoGetDeviceProperty** returns STATUS_BUFFER_TOO_SMALL, it sets this parameter to the required buffer length.

Include

wdm.h or *ntddk.h*

Return Value

IoGetDeviceProperty returns STATUS_SUCCESS if the call was successful. Possible error return values include:

Error Status	Description
STATUS_BUFFER_TOO_SMALL	The buffer at *PropertyBuffer* was too small. *ResultLength* points to the required buffer length.
STATUS_INVALID_PARAMETER_2	The given *DeviceProperty* is not one of the properties handled by this routine.
STATUS_INVALID_DEVICE_REQUEST	Possibly indicates that the given *DeviceObject* was not a valid PDO pointer.

Comments

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.

For many *DeviceProperty* requests, it can take two or more calls to **IoGetDeviceProperty** to determine the required *BufferLength*. The first call should use a best-guess value. If the return status is STATUS_BUFFER_TOO_SMALL, the driver should free its current buffer, allocate a buffer of the size returned in *ResultLength*, and call **IoGetDeviceProperty** again. Because some of the setup properties are dynamic, the data size can change between the time the required size is returned and driver calls this routine again. Therefore, drivers should call **IoGetDeviceProperty** inside a loop that executes until the return status is not STATUS_BUFFER_TOO_SMALL.

Function drivers that support devices on a legacy bus and a PnP bus can use the **DevicePropertyBusNumber**, **DevicePropertyBusTypeGuid**, and **DevicePropertyLegacyBusType** properties to distinguish between the buses.

Callers of **IoGetDeviceProperty** must be running at IRQL PASSIVE_LEVEL in the context of a system thread.

See Also

ExAllocatePool, **ExAllocatePoolWithTag**, CM_RESOURCE_LIST, IO_RESOURCE_REQUIREMENTS_LIST, GUID

IoInitializeRemoveLock

```
VOID
  IoInitializeRemoveLock(
    IN PIO_REMOVE_LOCK Lock,
    IN ULONG AllocateTag,
    IN ULONG MaxLockedMinutes,
    IN ULONG HighWatermark
    );
```

IoInitializeRemoveLock initalizes a remove lock for a device object. A driver can use the lock to track outstanding I/O on a device and to determine when the driver can delete its device object in response to an IRP_MN_REMOVE_DEVICE request.

Parameters

Lock

Points to a caller-supplied IO_REMOVE_LOCK structure that this routine initializes with information about the lock, including a counter and a synchronization event. A driver writer must allocate this structure as part of the device object's device extension.

AllocateTag

Specifies a tag to identify the creator of the lock. Driver writers typically use a 4-character string, specified in reverse order, like the tags used for **ExAllocatePoolWithTag**.

The I/O system only uses this parameter on checked builds.

MaxLockedMinutes

Specifies the maximum number of minutes that this lock should be held. A value of zero means there is no limit. This value is typically used during debugging to identify a driver routine that holds the lock longer than expected.

The I/O system only uses this parameter on checked builds. If the lock is held for more than *MaxLockedMinutes* on a checked build, the system asserts.

HighWatermark

Specifies the maximum number of outstanding acquisitions allowed on the lock.

The I/O system only uses this parameter on checked builds. If the lock is acquired *HighWatermark* times on a checked build, the system asserts.

Include

ntddk.h

Comments

The **Io***Xxx***RemoveLock***Xxx* routines provide a way to track the number of outstanding I/O's on a device and determine when it is safe to detach and delete a driver's device object. The system provides these routines to driver writers as an alternative to implementing their own tracking mechanism.

1. To ensure that the driver's DispatchPnP routine will not complete an IRP_MN_REMOVE_DEVICE request while the lock is held (for example, while another driver routine is accessing the device).
2. To count the number of reasons why the driver should not delete its device object, and to set an event when that count goes to zero.

A driver typically calls **IoInitializeRemoveLock** in its AddDevice routine, when the driver initializes the rest of the device extension for a device object.

A driver calls **IoAcquireRemoveLock** each time it starts an I/O operation. A driver calls **IoReleaseRemoveLock** each time it finishes an I/O operation. A driver can acquire the lock more than once; the **Io***Xxx***RemoveLock***Xxx* routines maintain a count of the outstanding acquisitions of the lock.

A driver should also call **IoAcquireRemoveLock** when it passes out a reference to its code (for timers, DPCs, callbacks, etc.). The driver calls **IoReleaseRemoveLock** when the event has returned.

In its dispatch code for IRP_MN_REMOVE_DEVICE, a driver acquires the lock once more and calls **IoReleaseRemoveLockAndWait**. This routine causes the driver to block until all outstanding acquisitions of the lock have been released. A driver should call **Io-ReleseRemoveLockAndWait** after it passes the remove IRP to the next-lower driver but before it releases memory, calls **IoDetachDevice**, or calls **IoDeleteDevice**.

A driver stores the IO_REMOVE_LOCK structure in the device extension of a device object. The remove lock is deleted when the driver deletes the device extension as part of processing an IRP_MN_REMOVE_DEVICE request.

Callers of **IoInitializeRemoveLock** must be running at IRQL = PASSIVE_LEVEL.

See Also

IoAcquireRemoveLock, **IoReleaseRemoveLock**, **IoReleaseRemoveLockAndWait**

IoInitializeRemoveLockEx

This routine is reserved for system use. See **IoInitializeRemoveLock**.

IoInvalidateDeviceRelations

```
VOID
  IoInvalidateDeviceRelations(
    IN PDEVICE_OBJECT DeviceObject,
    IN DEVICE_RELATION_TYPE Type
    );
```

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.

Parameters

DeviceObject

Points to the PDO for the device.

Type

Specifies the type of relations that have changed. Possible values include **BusRelations**, **EjectionRelations**, **RemovalRelations**, and **TargetDeviceRelation**.

Include

wdm.h or *ntddk.h*

Comments

For some relation types, such as **BusRelations**, this routine causes the PnP or Power Manager to gather updated relations information by sending an IRP_MN_QUERY_DEVICE_RELATIONS request to the drivers for the device. For other relation types, such as **EjectionRelations**, the PnP Manager does not need to gather new relation information immediately; the PnP Manager queries drivers for ejection relations only when it is preparing to eject a device.

After a bus driver calls **IoInvalidateDeviceRelations** to inform the PnP Manager that a device has disappeared, the bus driver must continue to handle PnP IRPs for that device until it receives an IRP_MN_REMOVE_DEVICE. In response to such IRPs, the bus driver returns STATUS_NO_SUCH_DEVICE. Until it succeeds the remove IRP, the bus driver can access the device extension to check its flags for the device.

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

See Also

IRP_MN_QUERY_DEVICE_RELATIONS

IoInvalidateDeviceState

```
VOID
  IoInvalidateDeviceState(
    IN PDEVICE_OBJECT PhysicalDeviceObject
    );
```

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.

Parameters

PhysicalDeviceObject

Points to the PDO for the device.

Include

wdm.h or *ntddk.h*

Comments

Drivers call this routine to indicate that something has changed with respect to one of the following aspects of a device's PnP state:

PNP_DEVICE_DISABLED
PNP_DEVICE_DONT_DISPLAY_IN_UI
PNP_DEVICE_FAILED
PNP_DEVICE_NOT_DISABLEABLE
PNP_DEVICE_REMOVED
PNP_DEVICE_RESOURCE_REQUIREMENTS_CHANGED

In response to this routine, the PnP Manager sends an IRP_MN_QUERY_PNP_DEVICE_STATE request to the device stack, to determine the current PnP state of the device.

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

See Also

IRP_MN_QUERY_PNP_DEVICE_STATE, PNP_DEVICE_STATE

IoOpenDeviceInterfaceRegistryKey

```
NTSTATUS
  IoOpenDeviceInterfaceRegistryKey(
    IN PUNICODE_STRING SymbolicLinkName,
    IN ACCESS_MASK DesiredAccess,
    OUT PHANDLE DeviceInterfaceKey
    );
```

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

Parameters

SymbolicLinkName

Points to a string identifying the device interface. This string was obtained from a previous call to **IoGetDeviceInterfaces**, **IoGetDeviceInterfaceAlias**, or **IoRegisterDeviceInterface**.

DesiredAccess

Specifies the access the caller requires to the key, such as KEY_READ, KEY_WRITE, or KEY_ALL_ACCESS.

DeviceInterfaceKey

Points to a returned handle to the requested registry key if the call is successful.

Include

wdm.h or *ntddk.h*

Return Value

IoOpenDeviceInterfaceRegistryKey returns STATUS_SUCCESS if the call was successful. Possible error return values include:

Error Status	Description
STATUS_OBJECT_NAME_NOT_FOUND	The routine was not able to locate a registry key for the device interface, probably due to an error in the *SymbolicLinkName*.
STATUS_OBJECT_PATH_NOT_FOUND	The routine was not able to locate a registry key for the device interface, probably due to an error in the *SymbolicLinkName*.

Continued

Error Status	Description
STATUS_INVALID_PARAMETER	Possibly indicates an error in the *SymbolicLinkName*.

Comments

IoOpenDeviceInterfaceRegistryKey opens a non-volatile subkey of the registry key for the device interface specified by *SymbolicLinkName*. Drivers can store information in this subkey that is specific to this instance of the device interface, such as the default resolution for a camera. User-mode applications can access this subkey using **SetupDi***Xxx* routines.

The driver must call **ZwClose** to close the handle returned from this routine when access is no longer required.

Callers of **IoOpenDeviceInterfaceRegistryKey** must be running at IRQL PASSIVE_ LEVEL in the context of a system thread.

See Also

IoGetDeviceInterfaces, **IoGetDeviceInterfaceAlias**, **IoRegisterDeviceInterface**, **ZwClose**

IoOpenDeviceRegistryKey

```
NTSTATUS
  IoOpenDeviceRegistryKey(
    IN PDEVICE_OBJECT DeviceObject,
    IN ULONG DevInstKeyType,
    IN ACCESS_MASK DesiredAccess,
    OUT PHANDLE DevInstRegKey
    );
```

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

Parameters

DeviceObject

Points to the PDO of the device instance for which the registry key is to be opened.

DevInstKeyType

Specifies flags indicating whether to open a device-specific or a driver-specific key. The flags also indicate whether the key is relative to the current hardware profile. May be a combination of the following values:

PLUGPLAY_REGKEY_DEVICE

Open a key for storing device-specific information. The key is located under the key for the device instance specified by *DeviceObject*. This flag may not be specified with PLUGPLAY_REGKEY_DRIVER.

PLUGPLAY_REGKEY_DRIVER

Open a key for storing driver-specific information. This flag may not be specified with PLUGPLAY_REGKEY_DEVICE.

PLUGPLAY_REGKEY_CURRENT_HWPROFILE

Open a key relative to the current hardware profile for device or driver information. This allows the driver to access configuration information that is hardware-profile-specific. The caller must specify either PLUGPLAY_REGKEY_DEVICE or PLUGPLAY_REGKEY_DRIVER with this flag.

DesiredAccess

Specifies the access the caller needs to the key.

DevInstRegKey

Points to a caller-allocated buffer that, on successful return, contains a handle to the requested registry key.

Include

wdm.h or *ntddk.h*

Return Value

IoOpenDeviceRegistryKey returns STATUS_SUCCESS if the call was successful. Possible error return values include:

Error Status	Description
STATUS_INVALID_PARAMETER	Possibly indicates that the caller specified an illegal set of *DevInstKeyType* flags.
STATUS_INVALID_DEVICE_REQUEST	Possibly indicates that the *DeviceObject* is not a valid PDO.

Comments

The driver must call **ZwClose** to close the handle returned from this routine when access is no longer required.

The registry keys opened by this routine are non-volatile.

User-mode configuration utilities, such as Class Installers, can access these same registry keys using the configuration manager and device installer APIs.

Callers of **IoOpenDeviceRegistryKey** must be running at IRQL PASSIVE_LEVEL in the context of a system thread.

See Also

ZwClose

IoRegisterDeviceInterface

```
NTSTATUS
  IoRegisterDeviceInterface(
    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN CONST GUID *InterfaceClassGuid,
    IN PUNICODE_STRING ReferenceString  OPTIONAL,
    OUT PUNICODE_STRING SymbolicLinkName
    );
```

IoRegisterDeviceInterface registers device functionality (a device interface) that a driver will enable for use by applications or other system components.

Parameters

PhysicalDeviceObject

Points to the PDO for the device.

InterfaceClassGuid

Points to the class GUID that identifies the functionality (the device interface) being registered.

ReferenceString

Optionally points to a reference string. Function drivers typically specify NULL for this parameter. Filter drivers must specify NULL.

Reference strings are only used by a few bus drivers that use device interfaces as placeholders for software devices that are created on demand. The reference string for a device interface is passed to the driver by the I/O Manager when the interface is opened. The string becomes part of the interface's name (as an additional path component). The driver uses reference strings to differentiate between two interfaces of the same class for a single device.

On Microsoft® Windows® 98 systems, the *ReferenceString* can be no longer than MAX_PATH characters. There is no length limit on Windows 2000 systems.

SymbolicLinkName

Points to a unicode string structure allocated by the caller. If this routine is successful, it initializes the unicode string and allocates the string buffer containing the kernel-mode path to the symbolic link for this device interface.

The caller must treat *SymbolicLinkName* as opaque and must not disassemble it.

The caller is responsible for freeing *SymbolicLinkName* with **RtlFreeUnicodeString** when it is no longer needed.

Include

wdm.h or *ntddk.h*

Return Value

IoRegisterDeviceInterface returns STATUS_SUCCESS if the call was successful. Possible error return values include:

Error Status	Description
STATUS_INVALID_DEVICE_REQUEST	Possibly indicates that the *PhysicalDeviceObject* is not a valid PDO pointer.

Comments

IoRegisterDeviceInterface registers a device interface and returns the name of the interface. A driver can call this routine several times for a given device to register several interfaces. A function or filter driver typically registers device interfaces in its AddDevice routine. For example, a fault-tolerant volume driver might register a fault-tolerant-volume interface and a volume interface for a particular volume.

The I/O Manager creates a registry key for the device interface. Drivers can access persistent storage under this key using **IoOpenDeviceInterfaceRegistryKey**.

A driver registers an interface once and then calls **IoSetDeviceinterfaceState** to enable and disable the interface.

If the device interface specified by the *PhysicalDeviceObject*, *InterfaceClassGuid*, and optional *ReferenceString* already exists, this routine returns STATUS_SUCCESS and the *SymbolicLinkName* for the existing interface.

Most drivers use a NULL reference string for a device interface. If a driver uses a non-NULL reference string, it must do additional work including possibly managing its own namespace and security. A filter driver that exposes a device interface must use a NULL *ReferenceString* to avoid conflicts with other drivers in the device stack.

Callers of this routine are not required to remove the registration for a device interface when it is no longer needed. Device interface registrations can be removed from user mode, if necessary.

Callers of **IoRegisterDeviceInterface** must be running at IRQL PASSIVE_LEVEL in the context of a system thread.

See Also

IoGetDeviceInterfaces, **IoOpenDeviceInterfaceRegistryKey**, **IoSetDeviceinterfaceState**, **RtlFreeUnicodeString**

IoRegisterPlugPlayNotification

```
NTSTATUS
  IoRegisterPlugPlayNotification(
    IN IO_NOTIFICATION_EVENT_CATEGORY EventCategory,
    IN ULONG EventCategoryFlags,
    IN PVOID EventCategoryData  OPTIONAL,
    IN PDRIVER_OBJECT DriverObject,
    IN PDRIVER_NOTIFICATION_CALLBACK_ROUTINE CallbackRoutine,
    IN PVOID Context,
    OUT PVOID *NotificationEntry
    );
```

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

Parameters

EventCategory

Specifies the category of PnP event for which the callback routine is being registered. *EventCategory* must be one of the following:

EventCategoryDeviceInterfaceChange

PnP events in this category include the arrival (enabling) of a new device interface (GUID_ DEVICE_INTERFACE_ARRIVAL) or the removal (disabling) of an existing device interface (GUID_DEVICE_INTERFACE_REMOVAL). See **IoRegisterDeviceInterface** for more information on device interfaces.

EventCategoryHardwareProfileChange

PnP events in this category include query-change (GUID_HWPROFILE_QUERY_ CHANGE), change-complete (GUID_HWPROFILE_CHANGE_COMPLETE), and change-cancel (GUID_HWPROFILE_CHANGE_CANCELLED) of a hardware profile.

EventCategoryTargetDeviceChange

PnP events in this category include events related to removing a device: the device's drivers received a query-remove IRP (GUID_TARGET_DEVICE_QUERY_REMOVE), the drivers completed a remove IRP (GUID_TARGET_DEVICE_REMOVE_COMPLETE), or the drivers received a cancel-remove IRP (GUID_TARGET_DEVICE_REMOVE_CANCELLED). This category is also used for custom notification events.

EventCategoryFlags

Specifies flags that modify the registration operation. Possible values include:

PNPNOTIFY_DEVICE_INTERFACE_INCLUDE_EXISTING_INTERFACES

Only valid with an *EventCategory* of **EventCategoryDeviceInterfaceChange**. If set, the PnP Manager calls the driver callback routine for each device interface that is currently registered and active and registers the callback routine for future device interface arrivals or removals.

EventCategoryData

Points to further information about the events for which *CallbackRoutine* is being registered. The information varies for different *EventCategory* values:

- When *EventCategory* is **EventCategoryDeviceInterfaceChange**, *EventCategoryData* must point to a GUID specifying a device interface class. *CallbackRoutine* will be called when an interface of that class is enabled or removed.
- When *EventCategory* is **EventCategoryHardwareProfileChange**, *EventCategoryData* must be NULL.
- When *EventCategory* is **EventCategoryTargetDeviceChange**, *EventCategoryData* must point to the file object for which PnP notification is requested.

DriverObject

Points to the caller's driver object.

To ensure that the driver remains loaded while it is registered for PnP notification, this call increments the reference count on *DriverObject*. The PnP Manager decrements the reference count when this registration is removed.

CallbackRoutine

Points to the routine to be called when the specified PnP event occurs.

A callback routine has the following type:

```
typedef NTSTATUS (*PDRIVER_NOTIFICATION_CALLBACK_ROUTINE) (
    IN PVOID NotificationStructure,
    IN PVOID Context
    );
```

The *NotificationStructure* is specific to the *EventCategory*. For example, a callback routine for an **EventCategoryDeviceInterfaceChange** receives a DEVICE_INTERFACE_CHANGE_NOTIFICATION structure.

The *Context* parameter contains the context data the driver supplied during registration.

The PnP Manager calls driver callback routines at IRQL PASSIVE_LEVEL.

Context

Points to a caller-allocated buffer containing context that the PnP Manager passes to the callback routine.

NotificationEntry

Points to an opaque value returned by this call that identifies the registration. Pass this value to **IoUnregisterPlugPlayNotification** to remove the registration.

Include

wdm.h or *ntddk.h*

Return Value

IoRegisterPlugPlayNotification returns STATUS_SUCCESS or an appropriate error status.

Comments

A driver registers for an event category. Each category includes one or more PnP events.

A driver can register different callback routines for different event categories or can register a single callback routine. A single callback routine can cast the *NotificationStructure* to a PLUGPLAY_NOTIFICATION_HEADER and use the **Event** field to determine the exact type of the notification structure.

Notification callback routines should complete their tasks as quickly as possible and return control to the PnP Manager, to prevent delays in notifying other drivers and applications that have registered for the event.

The PnP Manager does not take out a reference on the file object when a driver registers for notification of an **EventCategoryTargetDeviceChange**. If the driver's notification callback routine requires access to the file object, the driver should take out an extra reference on the file object before calling **IoRegisterPlugPlayNotification**.

See the *Plug & Play, Power Management, and Setup Design Guide* for more information on using PnP notification.

Callers of **IoRegisterPlugPlayNotification** must be running at IRQL PASSIVE_LEVEL.

See Also

DEVICE_INTERFACE_CHANGE_NOTIFICATION, HWPROFILE_CHANGE_NOTIFICATION, **IoUnregisterPlugPlayNotification**, PLUGPLAY_NOTIFICATION_HEADER, TARGET_DEVICE_CUSTOM_NOTIFICATION, TARGET_DEVICE_REMOVAL_NOTIFICATION

IoReleaseRemoveLock

```
VOID
  IoReleaseRemoveLock(
    IN PIO_REMOVE_LOCK RemoveLock,
    IN PVOID Tag
    );
```

IoReleaseRemoveLock releases a remove lock acquired with a previous call to **IoAcquireRemoveLock**.

Parameters

RemoveLock

Points to an IO_REMOVE_LOCK structure that the caller passed to a previous call to **IoAcquireRemoveLock**.

Tag

Points to a caller-supplied tag that was passed to a previous call to **IoAcquireRemoveLock**.

If a driver specified a *Tag* when it acquired the lock, the driver must specify the same *Tag* when releasing the lock. If the tags do not match, this routine asserts on a checked build.

If the call to **IoAcquireRemoveLock** did not specify a *Tag*, then this parameter is NULL.

The I/O system only uses this parameter on checked builds.

Include

ntddk.h

Comments

Each call to **IoAcquireRemoveLock** must have a corresponding call to **IoRelease-RemoveLock**.

IoReleaseRemoveLock decrements the count of outstanding acquisitions of the remove lock. If the count goes to zero and the driver has received an IRP_MN_REMOVE_DEVICE request, **IoReleaseRemoveLock** sets the event that allows the driver's remove dispatch code to detach and delete the device object. **IoReleaseRemoveLock** does not delete the lock; it decrements the count.

A driver calls a similar routine, **IoReleaseRemoveLockAndWait**, only in its dispatch code for an IRP_MN_REMOVE_DEVICE request. A driver calls **IoReleaseRemoveLockAnd-Wait** to ensure that all outstanding locks have been released before it detaches and deletes the device object.

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

See Also

IoAcquireRemoveLock, **IoInitializeRemoveLock**, **IoReleaseRemoveLockAndWait**

IoReleaseRemoveLockEx

This routine is reserved for system use. See **IoReleaseRemoveLock**.

IoReleaseRemoveLockAndWait

```
VOID
  IoReleaseRemoveLockAndWait(
    IN PIO_REMOVE_LOCK RemoveLock,
    IN PVOID Tag
    );
```

IoReleaseRemoveLockAndWait releases a remove lock acquired with a previous call to **IoAcquireRemoveLock** and waits until all acquisitions of the lock have been released. A driver typically calls this routine in its dispatch code for an IRP_MN_REMOVE_DEVICE request.

Parameters

RemoveLock

Points to an IO_REMOVE_LOCK structure that the caller passed to a previous call to **IoAcquireRemoveLock**.

Tag

Points to a caller-supplied tag that was passed to a previous call to **IoAcquireRemoveLock**.

If a driver specified a *Tag* when it acquired the lock, the driver must specify the same *Tag* when releasing the lock. If the tags do not match, this routine asserts on a checked build.

If the call to **IoAcquireRemoveLock** did not specify a *Tag*, then this parameter is NULL.

The I/O system only uses this parameter on checked builds.

Include

ntddk.h

Comments

A driver typically calls this routine in its dispatch code for an IRP_MN_REMOVE_DEVICE request. A driver should call **IoReleseRemoveLockAndWait** after it passes the remove IRP to the next-lower driver but before it releases memory, calls **IoDetachDevice**, or calls **IoDeleteDevice**.

A driver must acquire the remove lock once more before calling **IoReleaseRemoveLockAndWait**. Typically, a driver calls **IoAcquireRemoveLock** early in its DispatchPnp routine, before the switch statement. Then the lock is acquired for each PnP operation, including the acquisition required before calling **IoReleaseRemoveLockAndWait** in the code that handles the IRP_MN_REMOVE_DEVICE.

To release a lock from code other than the IRP_MN_REMOVE_DEVICE dispatch code, use **IoReleaseRemoveLock**.

Callers of **IoReleaseRemoveLockAndWait** must be running at IRQL PASSIVE_LEVEL.

See Also

IoAcquireRemoveLock, **IoInitializeRemoveLock**, **IoReleaseRemoveLock**

IoReleaseRemoveLockAndWaitEx

This routine is reserved for system use. See **IoReleaseRemoveLockAndWait**.

IoReportDetectedDevice

```
NTSTATUS
  IoReportDetectedDevice(
    IN PDRIVER_OBJECT DriverObject,
    IN INTERFACE_TYPE LegacyBusType,
    IN ULONG BusNumber,
    IN ULONG SlotNumber,
    IN PCM_RESOURCE_LIST ResourceList,
    IN PIO_RESOURCE_REQUIREMENTS_LIST ResourceRequirements  OPTIONAL,
    IN BOOLEAN ResourceAssigned,
    IN OUT PDEVICE_OBJECT *DeviceObject
    );
```

IoReportDetectedDevice reports a nonPnP device to the PnP Manager.

Parameters

DriverObject

Points to the driver object of the driver that detected the device.

LegacyBusType

Specifies the type of bus on which the device resides. The PnP Manager uses this information to match the reported device to its PnP-enumerated instance, if one exists.

The interface types, such as **PCIBus**, are defined in *ntddk.h*. If a driver does not know the *LegacyBusType* for the device, the driver supplies the value **InterfaceTypeUndefined** for this parameter.

BusNumber

Specifies the bus number for the device. The PnP Manager uses this information to match the reported device to its PnP-enumerated instance, if one exists.

The bus number distinguishes the bus on which the device resides from other buses of the same type on the machine. The bus-numbering scheme is bus-specific. If a driver does not know the *BusNumber* for the device, the driver supplies the value -1 for this parameter.

SlotNumber

Specifies the logical slot number of the device. The PnP Manager uses this information to match the reported device to its PnP-enumerated instance, if one exists.

If a driver does not know the *SlotNumber* for the device, the driver supplies the value -1 for this parameter.

ResourceList

Points to the resource list the driver used to detect the device. Resources in this list are in raw, untranslated form.

ResourceRequirements

Optionally points to a resource requirements list for the detected device. NULL if the caller does not have this information for the device.

ResourceAssigned

Specifies whether the device's resources have already been reported to the PnP Manager. If *ResourceAssigned* is TRUE, the resources have already been reported, possibly with **IoReportResourceForDetection**, and the PnP Manager will not attempt to claim them on behalf of the device. If TRUE, the PnP Manager will also not claim resources when the device is root-enumerated on subsequent boots.

DeviceObject

Optionally points to a PDO for the detected device.

NULL if the caller does not have a PDO for the device, which is typically the case. If *DeviceObject* is NULL, the PnP Manager creates a PDO for the device and returns a pointer to the caller.

If the caller supplies a PDO, the PnP Manager does not create a new PDO. On a given call to this routine the *DeviceObject* parameter is either an IN or an OUT parameter, but not both.

Include

ntddk.h

Return Value

IoReportDetectedDevice returns STATUS_SUCCESS or an appropriate error status.

Comments

A driver should only call **IoReportDetectedDevice** to report a legacy, nonPnP device. All PnP devices should be enumerated in response to an IRP_MN_QUERY_DEVICE_RELATIONS request.

A driver must only do detection on a device if the *DoDetectionNow* flag is set in the registry. This flag is typically set in the Services subkey for the driver by an installer. After reporting detected devices the driver must clear the flag. The location of this flag is driver-defined.

A driver typically calls this routine from its **DriverEntry** routine. A few drivers, like certain NDIS or EISA drivers, might call this routine from an AddDevice routine.

On successful completion of **IoReportDetectedDevice**, the caller should attach an FDO to the PDO returned at *DeviceObject*. Once the caller attaches its FDO, the caller is the function driver for the device, at least temporarily. There are no filter drivers. The PnP Manager owns the PDO.

The PnP Manager considers the device to be started and therefore does not call the driver's AddDevice routine and does not send an IRP_MN_START_DEVICE request. The driver must be prepared to handle all other PnP IRPs, however.

IoReportDetectedDevice marks the device as a root-enumerated device and this identification is persistent across system boots. During subsequent system boots the PnP Manager "detects" the device on the root-enumerated list and configures it like a PnP device: the PnP

Manager queries for device information, identifies the appropriate drivers and calls their AddDevice routines, and sends all the appropriate PnP IRPs. The driver that orignally detected the device may or may not be in the device stack on subsequent boots. It depends on the device's hardware ID and the resulting INF match, as is true when configuring any PnP device.

In certain situations the PnP Manager removes the reporting driver from the device stack and builds a full device stack without waiting for the system to reboot. (For example, when the user-mode PnP Manager detects the new device.) In such cases, the PnP Manager sends IRP_MN_QUERY_ID requests to determine the device's hardware ID and compatible IDs searches for an INF match. If it finds a match, it sends an IRP_MN_QUERY_REMOVE_DEVICE and an IRP_MN_REMOVE_DEVICE to the device stack to remove the existing drivers (which at this point are only the driver that called **IoReportDetectedDevice** and the parent bus driver). The PnP Manager then rebuilds the device stack using information in the INF file and the registry.

Callers of **IoReportDetectedDevice** must be running at IRQL PASSIVE_LEVEL in the context of a system thread.

See Also

IoReportResourceForDetection, IRP_MN_QUERY_DEVICE_RELATIONS

IoReportResourceForDetection

```
NTSTATUS
  IoReportResourceForDetection(
    IN PDRIVER_OBJECT DriverObject,
    IN PCM_RESOURCE_LIST DriverList    OPTIONAL,
    IN ULONG DriverListSize    OPTIONAL,
    IN PDEVICE_OBJECT DeviceObject    OPTIONAL,
    IN PCM_RESOURCE_LIST DeviceList    OPTIONAL,
    IN ULONG DeviceListSize    OPTIONAL,
    OUT PBOOLEAN ConflictDetected
    );
```

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.

Parameters

DriverObject

Points to the driver object that was input to the driver's **DriverEntry** routine.

DriverList

Optionally points to a caller-supplied buffer containing the driver's resource list, if the driver claims the same resources for all its devices. If the caller specifies a *DeviceList*, this parameter is ignored.

DriverListSize

Specifies the size in bytes of an optional *DriverList*. If *DriverList* is NULL, this parameter should be zero.

DeviceObject

Optionally points to the device object representing device for which the driver is attempting to claim resources.

DeviceList

Optionally points to a caller-supplied buffer containing the device's resource list. If the driver claims the same resources for all its devices, the caller can specify a *DriverList*.

DeviceListSize

Specifies the size in bytes of an optional *DeviceList*. If *DeviceList* is NULL, this parameter should be zero.

ConflictDetected

Points to a caller-supplied BOOLEAN value set to TRUE on return if the resources are not available.

Include

ntddk.h

Return Value

IoReportResourceForDetection returns STATUS_SUCCESS if the resources are claimed. Possible error return values include:

Error Status	Description
STATUS_CONFLICTING_ADDRESSES	The resources could not be claimed because they are in use or needed for a PnP-enumerable device.
STATUS_UNSUCCESSFUL	The *DeviceList* or *DriverList* is invalid.

Comments

If a driver supports only PnP hardware, it does no detection and therefore does not call **IoReportResourceForDetection**. The PnP system enumerates each PnP device, assigns resources to the device, and passes those resources to the device's driver(s) in an IRP_MN_START_DEVICE request.

If a PnP driver supports legacy hardware, however, it must call **IoReportResourceForDetection** to claim hardware resources before it attempts to detect the device.

Callers of this routine specify a CM_RESOURCE_LIST in either a *DeviceList* or a *DriverList*, allocated from paged memory. The caller is responsible for freeing the memory.

A driver that can control more than one legacy card at the same time should claim the resources for each device against the device object for the respective device (using a the *DeviceObject*, *DeviceList*, and *DeviceListSize* parameters). Such a driver must not claim these resources against their driver object.

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, the caller must allocate memory dynamically to contain the additional elements. 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.

IoReportResourceForDetection, with the help of the PnP Manager, determines whether the resources being requested conflict with resources that have already been claimed.

If a conflict is detected, this routine sets the BOOLEAN at *ConflictDetected* to TRUE and returns STATUS_CONFLICTING_ADDRESSES.

If no conflict is detected, this routine claims the resources, sets the BOOLEAN at *ConflictDetected* to FALSE, and returns STATUS_SUCCESS.

If this routine succeeds and the driver detects a legacy device, the driver reports the device to the PnP Manager by calling **IoReportDetectedDevice**. In that call, the driver sets *ResourceAssigned* to TRUE so the PnP Manager does not attempt to claim the resources again.

When a driver no longer requires the resources claimed by a call to this routine, the driver calls this routine again with a *DriverList* or *DeviceList* with a **Count** of zero.

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.

A driver can call this routine more than once for a given device. If one set of resources fails, the driver can call the routine again for the same device with a different set of resources. If a set of resources succeeds, the driver can call this routine again with a new list; the new list replaces the previous list.

Callers of **IoReportResourceForDetection** must be running at IRQL PASSIVE_LEVEL in the context of a system thread.

See Also

CM_RESOURCE_LIST, **IoReportDetectedDevice**

IoReportTargetDeviceChange

```
NTSTATUS
  IoReportTargetDeviceChange(
    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN PVOID NotificationStructure
    );
```

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. Do not use this routine to report system PnP events, such as GUID_TARGET_DEVICE_REMOVE_COMPLETE.

Parameters

PhysicalDeviceObject

Points to the PDO of the device being reported.

NotificationStructure

Points to a caller-supplied TARGET_DEVICE_CUSTOM_NOTIFICATION structure describing the custom event. The PnP Manager sends this structure to drivers that registered for notification of the event.

NotificationStructure.**FileObject** must be NULL. *NotificationStructure*.**Event** must contain the custom GUID for the event. The other fields of the *NotificationStructure* must be filled in as appropriate for the custom event.

The PnP Manager fills in the *NotificationStructure*.**FileObject** field when it sends notifications to registrants.

Include

wdm.h or *ntddk.h*

Return Value

IoReportTargetDeviceChange returns STATUS_SUCCESS or an appropriate error status. Possible error status values include:

Error Status	Description
STATUS_INVALID_DEVICE_REQUEST	The caller specified a system PnP event, such as GUID_TARGET_DEVICE_QUERY_REMOVE. This routine is only for custom events.

Comments

A driver that defines a custom device event calls **IoReportTargetDeviceChange** to inform the PnP Manager that the custom event has occurred. Custom notifcation can be used for events like a volume label change.

A driver should call the asynchronous form of this routine, **IoReportTargetDeviceChange-Asynchronous**, instead of this routine, to prevent deadlocks.

Certain kernel-mode components can call this synchronous routine. For example, a file system can call **IoReportTargetDeviceChange** to report a "get off the volume" custom event when a component tries to open the volume for exclusive access. Clients that register for notification on file system volumes are careful to not request an exclusive open in a PnP notification callback routine.

The custom notification structure contains a driver-defined event with its own GUID. Driver writers can generate GUIDs with *uuidgen.exe* or *guidgen.exe*.

Callers of **IoReportTargetDeviceChange** must be running at IRQL PASSIVE_LEVEL in the context of a system thread. To report a target device change from raised IRQL, call **Io-ReportTargetDeviceChangeAsynchronous**.

IoReportTargetDeviceChange is not supported on Windows 98; it returns STATUS_NOT_SUPPORTED.

See Also

IoReportTargetDeviceChangeAsynchronous, TARGET_DEVICE_CUSTOM_NOTIFICATION

IoReportTargetDeviceChangeAsynchronous

```
NTSTATUS
  IoReportTargetDeviceChangeAsynchronous(
    IN PDEVICE_OBJECT PhysicalDeviceObject,
    IN PVOID NotificationStructure,
    IN PDEVICE_CHANGE_COMPLETE_CALLBACK Callback    OPTIONAL,
    IN PVOID Context    OPTIONAL
    );
```

IoReportTargetDeviceChangeAsynchronous notifies the PnP Manager that a custom event has occurred on a device. The routine returns immediately; it does not wait while the PnP Manager sends notification of the event to drivers that registered for notification on the device. Do not use this routine to report system PnP events, such as GUID_TARGET_DEVICE_REMOVE_COMPLETE.

Parameters

PhysicalDeviceObject

Points to the PDO of the device being reported.

NotificationStructure

Points to a caller-supplied TARGET_DEVICE_CUSTOM_NOTIFICATION structure describing the custom event. The PnP Manager sends this structure to drivers that registered for notification of the event.

NotificationStructure.**FileObject** must be NULL. *NotificationStructure*.**Event** must contain the custom GUID for the event. The other fields of the *NotificationStructure* must be filled in as appropriate for the custom event.

The PnP Manager fills in the *NotificationStructure*.**FileObject** field when it sends notifications to registrants.

Callback

Optionally points to a caller-supplied routine that the PnP Manager calls after it finishes notifying drivers that registered for this custom event.

The callback routine has the following type:

```
typedef
VOID
(*PDEVICE_CHANGE_COMPLETE_CALLBACK)(
    IN PVOID Context
    );
```

A device-change-complete callback routine should not block and must not call synchronous routines that generate PnP events.

The PnP Manager calls device-change-complete callback routines at IRQL PASSIVE_LEVEL.

Context

Optionally points to a caller-supplied context structure that the PnP Manager passes to the *Callback* routine. The caller must allocate this structure from nonpaged memory.

Include

ntddk.h

Return Value

IoReportTargetDeviceChangeAsynchronous returns STATUS_SUCCESS or an appropriate error status. Possible error status values include:

Error Status	Description
STATUS_INVALID_DEVICE_REQUEST	The caller specified a system PnP event, such as GUID_TARGET_DEVICE_QUERY_REMOVE. This routine is only for custom events.

Comments

A driver that defines a custom device event calls **IoReportTargetDeviceChange-Asynchronous** to inform the PnP Manager that the custom event has occurred. Custom notification can be used for events like a volume label change.

The custom notification structure contains a driver-defined event with its own GUID. Driver writers can generate GUIDs with *uuidgen.exe* or *guidgen.exe*.

When a driver calls this routine while handling an event, an IRP_MN_REMOVE_DEVICE, or an IRP_MN_SURPRISE_REMOVAL, the PnP Manager calls the driver's *Callback* routine after the driver returns and the stack unwinds.

Callers of **IoReportTargetDeviceChangeAsynchronous** must be running at IRQL <= DISPATCH_LEVEL. If a driver writer calls this routine at IRQL DISPATCH_LEVEL, the *NotificationStructure* must be allocated from nonpaged memory.

See Also

IoReportTargetDeviceChange, TARGET_DEVICE_CUSTOM_NOTIFICATION

IoRequestDeviceEject

```
VOID
  IoRequestDeviceEject(
    IN PDEVICE_OBJECT PhysicalDeviceObject
    );
```

IoRequestDeviceEject notifies the PnP Manager that the device eject button was pressed. Note that this routine reports a request for device eject, not media eject.

Parameters

PhysicalDeviceObject

Points to the PDO for the device.

Include

wdm.h or *ntddk.h*

Comments

Typically, a PnP bus driver calls **IoRequestDeviceEject** to notify the PnP Manager that a user pressed the device eject button on one of its child devices.

A driver calls this routine, rather than sending an IRP_MN_EJECT request, because this routine allows the PnP Manager to coordinate additional actions for the eject besides sending the IRP. For example, the PnP Manager notifies user-mode and kernel-mode components that registered for notification of changes on the device.

The PnP Manager directs an orderly shutdown of the device. The PnP Manager:

1. Creates a list of other devices that are affected by this device being ejected.

   The PnP Manager queries for the device's removal relations, ejection relations, and bus relations (child devices).

2. Determines whether the device and its related devices can be software-removed.

   The PnP Manager sends IRP_MN_QUERY_REMOVE_DEVICE IRPs to the drivers for the device and its related devices. The PnP Manager also sends notifications to any user-mode and kernel-mode components that registered for device-change notification on the device or any of its related devices. If any of the drivers or user-mode components fail the query-remove, the PnP Manager pops up a dialog box to notify the user that the eject failed.

3. Software-removes the device and its related devices.

   If the previous steps are successful, the PnP Manager notifies registered drivers and applications that the device and its related devices are being software-removed. Then the PnP Manager sends IRP_MN_REMOVE_DEVICE IRPs to the drivers for the device and its related devices. Function and filter drivers detach from the device stack and delete their device objects for the device(s). The bus drivers retain the PDO(s) for the device(s), unless a device is physically gone and the bus driver omitted the device in its most recent response to IRP_MN_QUERY_DEVICE_RELATIONS/**BusRelations** for the device's parent bus.

4. Directs the bus driver to eject the device (if possible).

   The PnP Manager takes different steps, depending on the eject capabilities of the device:

   - Hot eject is supported.

     If the EjectSupported capability is set for the device, the device can be ejected when the system is running (is in the PowerSystemWorking state). The PnP Manager sends an IRP_MN_EJECT request to the bus driver for the device. Any function and filter drivers detached previously from the stack in response to the remove IRP, so the bus driver handles the eject IRP. When the bus driver completes the IRP, the PnP Manager expects the device to be physically absent from the system.

- Hot eject is not supported.

  In this case, the device is **Removable** but does not support eject. The PnP Manager marks the device as not-present/not-working-properly. The PnP Manager will not restart the device until a user physically removes it and reinserts it. In this case, the PnP Manager does not send an IRP_MN_EJECT.

A device's parent bus driver sets the capabilities for a device, including its eject capabilities, in response to an IRP_MN_QUERY_CAPABILITIES request. A function or filter driver can optionally specify capabilities.

When a device is ejected, its child devices are physically removed from the system along with it.

A user-mode application can initiate a device eject. In that case, no driver calls this routine but the OS calls the PnP Manager to initiate the steps listed above.

Callers of **IoRequestDeviceEject** must be running at IRQL <= DISPATCH_LEVEL. The PnP Manager performs most of its device-eject tasks listed above at IRQL PASSIVE_LEVEL.

See Also

IRP_MN_EJECT, IRP_MN_QUERY_REMOVE_DEVICE, IRP_MN_QUERY_DEVICE_RELATIONS, IRP_MN_REMOVE_DEVICE

IoSetDeviceInterfaceState

```
NTSTATUS
  IoSetDeviceInterfaceState(
    IN PUNICODE_STRING SymbolicLinkName,
    IN BOOLEAN Enable
    );
```

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

Parameters

SymbolicLinkName

Points to a string identifying the device interface being enabled or disabled. This string was obtained from a previous call to **IoRegisterDeviceInterface** or **IoGetDeviceInterfaces**.

Enable

TRUE indicates that the device interface is being enabled. FALSE indicates that the device interface is being disabled.

Include

wdm.h or *ntddk.h*

Return Value

IoSetDeviceInterfaceState returns STATUS_SUCCESS if the call was successful. This routine returns an informational status of STATUS_OBJECT_NAME_EXISTS if the caller requested to enable a device interface that was already enabled. Possible error return values include:

Error Status	Description
STATUS_OBJECT_NAME_NOT_FOUND	The caller tried to disable a device interface that was not enabled.

Comments

IoSetDeviceInterfaceState enables a registered device interface for use by applications and other system components. The interface must have been previously registered with **Io-RegisterDeviceInterface** or from user mode.

IoSetDeviceInterfaceState creates a symbolic link for a device interface that is being enabled.

A function or filter driver typically calls this routine with *Enable* set to TRUE after it successfully starts a device in response to an IRP_MN_START_DEVICE. Such a driver should disable the device interface (*Enable* equals FALSE) when it removes the device in response to an IRP_MN_REMOVE_DEVICE.

If a call to this routine successfully exposes a device interface, the system notifies any components that registered for PnP notification of a device class change. Similarly, if a call to this routine disables an existing device interface, the system sends appropriate notifications.

The PnP Manager does not send notification of device-interface arrivals until the IRP_MN_START_DEVICE IRP completes, indicating that all the drivers for the device have completed their start operations. In addition, the PnP Manager fails create requests for the device until the IRP_MN_START_IRP completes.

Callers of **IoSetDeviceInterfaceState** must be running at IRQL PASSIVE_LEVEL in the context of a system thread.

See Also

IoGetDeviceInterfaces, **IoRegisterDeviceInterface**, **IoRegisterPlugPlayNotification**

IoUnregisterPlugPlayNotification

```
NTSTATUS
  IoUnregisterPlugPlayNotification(
    IN PVOID NotificationEntry
    );
```

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

Parameters

NotificationEntry

Points to an opaque value representing the registration to be removed. The value was returned by a previous call to **IoRegisterPlugPlayNotification**.

Include

wdm.h or *ntddk.h*

Return Value

IoUnregisterPlugPlayNotification returns STATUS_SUCCESS if the registration was successfully removed.

Comments

IoUnregisterPlugPlayNotification removes one PnP notification registration; that is, the registration of one driver callback routine for one PnP event category.

Drivers should unregister a notification first, then free any related context buffer.

A driver cannot be unloaded until it removes all of its PnP notification registrations because there is a reference on its driver object for each active registration.

Callers of **IoUnregisterPlugPlayNotification** must be running at IRQL PASSIVE_LEVEL in the context of a system thread.

See Also

IoRegisterPlugPlayNotification

C H A P T E R 2

Plug and Play IRPs

This chapter describes the PnP IRPs that are sent to drivers. All PnP IRPs have the major code IRP_MJ_PNP and a minor code indicating the particular PnP request.

This chapter provides reference information for the individual IRPs. See the *Plug and Play, Power Management, and Setup Design Guide* for a description of the order in which the IRPs are sent, a discussion of how to handle IRPs in a DispatchPnp routine, and a general discussion of PnP concepts and terminology.

For each IRP and each kind of driver, a driver is either *required* to handle the IRP, can *optionally* handle the IRP, or *must not* handle the IRP. Consult the table below to identify which IRPs your driver will handle and then consult the reference pages for information on the individual IRPs. The IRPs are listed in functional order in the table and in alphabetical order in the IRP reference pages.

If an IRP is marked "No" in the table for a particular driver, that driver must not handle the IRP. The driver must pass the IRP to the next driver in the device stack as described in the reference page for the IRP.

The PnP Manager sends these IRPs. PnP drivers can send some of these IRPs, but only those so noted in this chapter.

The following are the PnP IRPs and the drivers that handle them:

PnP IRP	**Function or Filter Driver for Nonbus Device**	**Function Driver for Bus Device (for bus FDO)**	**Bus Driver or Bus Filter Driver (for child PDOs)**
IRP_MN_START_DEVICE	Required	Required	Required
IRP_MN_QUERY_STOP_DEVICE	Required	Required	Required
IRP_MN_STOP_DEVICE	Required	Required	Required
IRP_MN_CANCEL_STOP_DEVICE	Required	Required	Required
IRP_MN_QUERY_REMOVE_DEVICE	Required	Required	Required

Continued

PnP IRP	Function or Filter Driver for Nonbus Device	Function Driver for Bus Device (for bus FDO)	Bus Driver or Bus Filter Driver (for child PDOs)
IRP_MN_REMOVE_DEVICE	Required	Required	Required
IRP_MN_CANCEL_REMOVE_DEVICE	Required	Required	Required
IRP_MN_SURPRISE_REMOVAL	Required	Required	Required
IRP_MN_QUERY_CAPABILITIES	Optional	Optional	Required
IRP_MN_QUERY_PNP_DEVICE_STATE	Optional	Optional	Optional
IRP_MN_FILTER_RESOURCE_ REQUIREMENTS	Optional (1)	Optional (1)	No
IRP_MN_DEVICE_USAGE_ NOTIFICATION	Required (1)	Required (1)	Required (1)
IRP_MN_QUERY_DEVICE_RELATIONS			
BusRelations	Optional (1)	Required	No (2)
EjectionRelations	No	No	Optional
RemovalRelations	Optional	Optional	No
TargetDeviceRelation	No	No	Required
IRP_MN_QUERY_RESOURCES	No	No	Required (1)
IRP_MN_QUERY_RESOURCE_ REQUIREMENTS	No	No	Required (1)
IRP_MN_QUERY_ID			
BusQueryDeviceID	No	No	Required
BusQueryHardwareIDs	No	No	Optional
BusQueryCompatibleIDs	No	No	Optional
BusQueryInstanceID	No	No	Optional
IRP_MN_QUERY_DEVICE_TEXT	No	No	Required (1)
IRP_MN_QUERY_BUS_INFORMATION	No	No	Required (1)
IRP_MN_QUERY_INTERFACE	Optional	Optional	Required (1)
IRP_MN_READ_CONFIG	No	No	Required (1)
IRP_MN_WRITE_CONFIG	No	No	Required (1)
IRP_MN_EJECT	No	No	Required (1)
IRP_MN_SET_LOCK	No	No	Required (1)

(1) Required or Optional in certain situations. See the reference page for the IRP for more details.

(2) Bus filter drivers might handle a query for **BusRelations**.

IRP_MN_CANCEL_REMOVE_DEVICE

All PnP drivers must handle this IRP.

When Sent

The PnP Manager sends this IRP to inform the drivers for a device that the device will not be removed.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input

None

Output

None

I/O Status Block

A driver must set **Irp->IoStatus.Status** to STATUS_SUCCESS for this IRP. If a driver fails this IRP, the device is left in an inconsistent state.

Operation

This IRP must be handled first by the parent bus driver for a device and then by each higher driver in the device stack.

In response to this IRP, drivers return the device to the state it was in prior to receiving the IRP_MN_QUERY_REMOVE_DEVICE request.

If the device is already started when the driver receives this IRP, the driver simply sets status to success and passes the IRP to the next driver (or completes the IRP if the driver is a bus driver). For such a cancel-remove IRP, a function or filter driver need not set a completion routine. The device may not be in the remove-pending state, because, for example, the driver failed the previous IRP_MN_QUERY_REMOVE_DEVICE.

The PnP Manager calls any **EventCategoryTargetDeviceChange** notification callbacks with GUID_TARGET_DEVICE_REMOVE_CANCELLED *after* the IRP_MN_CANCEL_REMOVE_DEVICE request completes. Such callbacks were registered on the device by calling **IoRegisterPlugPlayNotification**. The PnP Manager also calls any user-mode components that registered for notification on the device by calling **RegisterDevice-Notification**.

If a file system is mounted on the device, it must undo any operations it did in response to the query-remove notification.

See the *Plug and Play, Power Management, and Setup Design Guide* for detailed information on handling remove IRPs and for the general rules for handling all PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

See Also

IoRegisterPlugPlayNotification, IRP_MN_QUERY_REMOVE_DEVICE

IRP_MN_CANCEL_STOP_DEVICE

All PnP drivers must handle this IRP.

When Sent

The PnP Manager sends this IRP, at some point after an IRP_MN_QUERY_STOP_DEVICE, to inform the drivers for a device that the device will not be stopped for resource reconfiguration.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input

None

Output

None

I/O Status Block

A driver must set **Irp->IoStatus.Status** to STATUS_SUCCESS for this IRP. If a driver fails this IRP, the device is left in an inconsistent state.

Operation

This IRP must be handled first by the parent bus driver for a device and then by each higher driver in the device stack.

In response to this IRP, drivers return the device to the started state. Drivers start any IRPs that were held while the device was in the stop-pending state.

If the device is already in an active state when the driver receives this IRP, a function or filter driver simply sets status to success and passes the IRP to the next driver. The parent bus driver completes the IRP. For such a cancel-stop IRP, a function or filter driver need not set a completion routine.

See the *Plug and Play, Power Management, and Setup Design Guide* for detailed information on handling stop IRPs and for the general rules for handling all PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

See Also

IRP_MN_QUERY_STOP_DEVICE

IRP_MN_DEVICE_USAGE_NOTIFICATION

System components send this IRP to ask the drivers for a device whether the device can support a special file. If all the drivers for the device succeed the IRP, the system creates the special file. The system also sends this IRP to inform drivers that a special file has been removed from the device. The special files can be a paging file, a crash dump file, or a hibernation file.

Function drivers must handle this IRP if their device can contain a paging file, dump file, or hibernation file. Filter drivers must handle this IRP if the function driver they are filtering handles the IRP. Bus drivers must handle this IRP for their adapter or controller (bus FDO) and for their child devices (child PDOs).

When Sent

The system sends this IRP when it is creating or deleting a paging file, dump file, or hibernation file. A driver can send this IRP to propagate device usage information to another device stack.

System components and drivers send this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

Parameters.UsageNotification.InPath is a BOOLEAN. When this parameter is TRUE, the system is creating a paging, crash dump, or hibernation file on the device. When **InPath** is FALSE, such a file has been removed from the device.

Parameters.UsageNotification.Type is an enum indicating the kind of file. This parameter has one of the following values: **DeviceUsageTypePaging**, **DeviceUsageTypeDumpFile**, or **DeviceUsageTypeHibernation**.

Output

None

I/O Status Block

Drivers set **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

Drivers do not modify the **Irp->IoStatus.Information** field; it remains at zero as set by the component sending the IRP.

Operation

A driver handles this IRP on the IRP's way down the device stack and on the IRP's way back up the stack.

A driver responds to this IRP with a procedure like the following:

- If **Parameters.UsageNotification.InPath** is TRUE, determine whether the device supports the special file.

  A driver should test for the specific **Parameters.UsageNotification.Type**(s) that the driver can support. Additional notification types might be added in the future.

  See further information below describing the actions required to support each notification type.

  If **Parameters.UsageNotification.InPath** is TRUE and the driver cannot support the special file on the device, the driver must complete the IRP with a failure status.

- If the device supports the special file:

  1. Take appropriate actions to reflect that the device now contains, or no longer contains, a special file.

     A driver typically increments or decrements a counter. For example, if **Parameters.UsageNotification.Type** is DeviceUsageTypePaging and **Parameters.UsageNotification.InPath** is TRUE, increment a count of the number of paging files on the device. Certain driver dispatch routines must check the counter(s).

     A device that contains a special file should not be disabled. A driver can call **IoInvalidateDeviceState**, requesting the PnP Manager to re-query for the device's PnP device state information. In response to the resulting IRP_MN_QUERY_PNP_DEVICE_STATE IRP, the driver should set the PNP_DEVICE_NOT_DISABLEABLE flag.

     If **InPath** is FALSE, a driver sets the DO_POWER_PAGABLE bit in its device object for the device.

2. Propagate the device usage information to any related devices that require the information.

   As part of its handling of an IRP_MN_DEVICE_USAGE_NOTIFICATION IRP, a driver might be required to pass the information to one or more other device stacks. Such a driver creates new IRP_MN_DEVICE_USAGE_NOTIFICATION IRP(s) and sends them to the appropriate device stack(s). The driver must wait for completion of any device-usage-notification IRP(s) it sends before the driver finishes processing the device-usage IRP it received.

   How to identify the related devices is device- and driver-specific. Typically, a driver sends the IRP to other drivers to which it would send I/O requests for the file. When a bus driver handles this request for a child device, it must send a usage notification IRP to the device stack for the device's parent.

   For example, when ftdisk is running a five-disk stripe set, it propagates paging, hibernate, and crash dump notifications to each of those five disks, since each of those devices can be required to handle paging, hibernate, or crash dump file operations.

3. In a function or filter driver, set an IoCompletion routine.

4. In a function or filter driver, set **Irp->IoStatus.Status** to STATUS_SUCCESS, set up the next stack location, and pass the IRP to the next lower driver with **IoCallDriver**. Do not complete the IRP.

   In a bus driver that is handling the IRP for a child PDO: set **Irp->IoStatus.Status** and complete the IRP (**IoCompleteRequest**).

5. During IRP completion processing:

   If an IoCompletion routine detects that a lower driver has failed the IRP, the function or filter driver must undo any operations it performed in response to the IRP and propagate the error. If the function or filter driver propagated the usage information to any other device stacks, the driver must send another usage IRP to those stacks to notify them of the failure.

   If status is STATUS_SUCCESS and **InPath** is TRUE, clear the DO_POWER_PAGABLE bit.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Supporting Paging, Crash Dump, and Hibernation Files on a Device

When any of a driver's special file counts is nonzero, the driver must support the presence of the special file(s) on its device (or a descendant device).

For a **DeviceUsageTypePaging** file created on its device, a driver must do the following:

- Lock code in memory for its DispatchRead, DispatchWrite, DispatchDeviceControl, and DispatchPower routines.
- Clear the DO_POWER_PAGABLE bit in its device object for the device (on the IRP's way up the device stack).
- Fail IRP_MN_QUERY_STOP_DEVICE and IRP_MN_QUERY_REMOVE_DEVICE requests for the device.

For a **DeviceUsageTypeDumpFile** file on its device, a driver must do the following:

- Lock code in memory for its DispatchRead, DispatchWrite, DispatchDeviceControl, and DispatchPower routines.
- Do not take the device out of the D0 state.

  Do not register the device for idle detection (**PoRegisterDeviceForIdleDetection**). If the device is already registered, cancel the registration. If the driver performs its own idle detection for the device, suspend such detection.
- Clear the DO_POWER_PAGABLE bit in its device object for the device (on the IRP's way up the device stack).
- Fail IRP_MN_QUERY_STOP_DEVICE and IRP_MN_QUERY_REMOVE_DEVICE requests for the device.

For a **DeviceUsageTypeHibernation** file on its device, a driver must do the following:

- Lock code in memory for its DispatchRead, DispatchWrite, DispatchDeviceControl, and DispatchPower routines.
- Ensure the device is in the D0 state when the driver receives an S4 system power IRP indicating that the system is about to hibernate.
- Do not power down the device in response to a D3 set-power IRP that is part of an S4 hibernate action.

  Upon receipt of such a D3 set-power IRP, perform all tasks required to put the device in the D3 state except for powering off the device and notifying the Power Manager (**PoSetPowerState**). The device must retain power until the hibernation file has been written.
- Clear the DO_POWER_PAGABLE bit in its device object for the device (on the IRP's way up the device stack).

- Fail IRP_MN_QUERY_STOP_DEVICE and IRP_MN_QUERY_REMOVE_DEVICE requests for the device.

See the *Plug and Play, Power Management, and Setup Design Guide* for more information about device power states, power IRPs, and supporting power management in drivers.

Sending This IRP

A driver can send an IRP_MN_DEVICE_USAGE_INFORMATION IRP, but only to propagate device usage information to another device stack. A driver is never the initial source of device usage information.

See Also

IoAdjustPagingPathCount, IRP_MN_QUERY_REMOVE_DEVICE, IRP_MN_QUERY_STOP_DEVICE

IRP_MN_EJECT

Bus drivers typically handle this request for their child devices (child PDOs) that support device ejection. Function and filter drivers do not receive this request.

When Sent

The PnP Manager sends this IRP to direct the appropriate driver or drivers to eject the device from its slot.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

None

Output

None

I/O Status Block

A bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a bus driver sets **Irp->IoStatus.Information** to zero.

If a bus driver does not handle this IRP, it leaves **Irp->IoStatus.Status** as is and completes the IRP.

Operation

For the device to be ejected, the device must be in the D3 device power state (off) and must be unlocked (if the device supports locking).

Any driver that returns success for this IRP must wait until the device has been ejected before completing the IRP.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Instead, see the reference page for the **IoRequestDeviceEject** routine.

See Also

IoRequestDeviceEject

IRP_MN_FILTER_RESOURCE_REQUIREMENTS

The PnP Manager sends this IRP to a device stack so the function driver can adjust the resources required by the device, if appropriate.

The function driver typically handles this IRP.

The parent bus driver (and bus filter drivers) should not handle this request for a child PDO; instead, such a driver should report resource requirements in response to an IRP_MN_QUERY_RESOURCE_REQUIREMENTS request.

Upper and lower filter drivers do not handle this IRP.

When Sent

The PnP Manager sends this IRP when it is preparing to allocate resource(s) to a device.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in the context of an arbitrary thread.

Input

Irp->IoStatus.Information points to an IO_RESOURCE_REQUIREMENTS_LIST containing the hardware resource requirements for the device. The pointer is NULL if the device consumes no hardware resources.

Parameters.FilterResourceRequirements.IoResourceRequirementList also points to an IO_RESOURCE_REQUIREMENTS_LIST, but the function driver should use the list in the **IoStatus** block.

Output

Returned in the I/O status block.

I/O Status Block

If a function driver handles this IRP, it handles it on the IRP's way back up the stack. If the function driver handles the IRP successfully, it sets **Irp->IoStatus.Status** to STATUS_ SUCCESS and sets **Irp->IoStatus.Information** to a pointer to an IO_RESOURCE_ REQUIREMENTS_LIST containing the filtered resource requirements. See the *Operation* section below for further information on setting the filtered resource list. If a function driver encounters an error when handling this IRP, it sets the error in **Irp->IoStatus.Status**. If a function driver does not handle this IRP, it uses **IoSkipCurrentIrpStackLocation** to pass the IRP down the stack unchanged.

Upper and lower-filter drivers do not handle this IRP. Such a driver calls **IoSkipCurrent-IrpStackLocation**, passes the IRP down to the next driver, must not modify **Irp->IoStatus**, and must not complete the IRP.

The parent bus driver does not handle this IRP. It leaves **Irp->IoStatus** as is and completes the IRP.

Operation

The PnP Manager sends an IRP_MN_QUERY_RESOURCE_REQUIREMENTS request to the parent bus driver for the device, before the function driver has attached its device object to the device stack. To give the function driver an opportunity to modify the device's resource requirements, if appropriate, the PnP Manager later sends an IRP_MN_FILTER_ RESOURCE_REQUIREMENTS request to the full device stack. The PnP Manager sends this IRP before it allocates hardware resources to the device during initial device configuration. The PnP Manager might also send this IRP during resource rebalancing.

When the PnP Manager sends this IRP, it supplies the driver stack with a resource requirements list, which drivers can modify and return. The PnP Manager supplies one of the following types of resource requirements list (listed in order of priority):

- Forced configuration (modified from a resource list to a resource requirements list)
- Override configuration
- Basic configuration
- Boot configuration (modified from a resource list to a resource requirements list)

If a function driver handles this IRP, it must set a completion routine and handle the IRP on its way back up the device stack. See the *Plug and Play, Power Management, and Setup Design Guide* for information on handling a PnP IRP on its way back up the device stack.

If the function driver is not changing the size of the current list pointed to by **Irp->Io-Status.Information**, the driver can modify the list in place. If the driver needs to change the size of the requirements list, the driver must allocate a new IO_RESOURCE_REQUIREMENTS_LIST list from paged memory and free the previous list. The PnP Manager frees the returned structure when it is no longer needed.

A function driver must preserve the order of resources in the list pointed to by **Irp->Io-Status.Information** and must not alter resource tags that it does not handle. The driver must take care to adjust the requirements list in a way that the device's parent bus supports. If a function driver adds a new resource to the requirements list, and that resource is assigned to the device, the function driver should filter that resource out of the IRP_MN_START_DEVICE before passing the start IRP down to the bus driver.

If the function driver for the device does not handle this IRP, the PnP Manager uses the resource requirements as specified by the parent bus driver in response to the IRP_MN_QUERY_RESOURCE_REQUIREMENTS request.

A function driver must be prepared to handle this IRP for a device at any time after the driver's AddDevice routine has been called for the device.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

See Also

ExAllocatePoolWithTag, **ExFreePool**, IO_RESOURCE_REQUIREMENTS_LIST, IRP_MN_START_DEVICE

IRP_MN_QUERY_BUS_INFORMATION

The PnP Manager uses this IRP to request the type and instance number of a device's parent bus.

Bus drivers should handle this request for their child devices (PDOs). Function and filter drivers do not handle this IRP.

When Sent

The PnP Manager sends this IRP when a device is enumerated.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

None

Output

Returned in the I/O status block.

I/O Status Block

A bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a bus driver sets **Irp->IoStatus.Information** to a pointer to a completed PNP_BUS_INFORMATION structure. (See the *Operation* section below for more information.) On an error, the bus driver sets **Irp->IoStatus.Information** to zero.

Function and filter drivers do not handle this IRP.

Operation

The information returned in response to this IRP is available to the function and filter drivers for devices on the bus. Function and filter drivers can call **IoGetDeviceProperty** to request a **DevicePropertyBusTypeGuid**, **DevicePropertyLegacyBusType**, or **DevicePropertyBusNumber**. Function and filter drivers that support devices on more than one bus can use this information to determine on which bus a particular device resides.

If a bus driver returns information in response to this IRP, it allocates a PNP_BUS_INFORMATION structure from paged memory. The PnP Manager frees the structure when it is no longer needed.

A PNP_BUS_INFORMATION structure has the following format:

```
typedef struct _PNP_BUS_INFORMATION {
    GUID BusTypeGuid;
    INTERFACE_TYPE LegacyBusType;
    ULONG BusNumber;
} PNP_BUS_INFORMATION, *PPNP_BUS_INFORMATION;
```

The members of the structure are defined as follows:

BusTypeGuid

A bus driver sets **BusTypeGuid** to the GUID for the type of the bus on which the device resides. GUIDs for standard bus types are listed in *wdmguid.h*. Driver writers should generate GUIDs for other bus types using **uuidgen**.

LegacyBusType

A PnP bus driver sets **LegacyBusType** to the INTERFACE_TYPE of the parent bus. The interface types are defined in *wdm.h*. Some buses have a specific INTERFACE_TYPE value, such as **PCMCIABus**, **PCIBus**, or **PNPISABus**. For other buses, especially newer buses like USB, the bus driver sets this member to **PNPBus**.

The **LegacyBusType** specifies the interface used to communicate with the device. This may or may not correspond to the type of the parent bus. For example, the interface for a Card-Bus card that is plugged into a PCI CardBus controller is **PCIBus**. However, the interface for a PCMCIA card on a PCI CardBus controller is **PCMCIABus**.

BusNumber

A bus driver sets **BusNumber** to a number distinguishing the bus from other buses of the same type on the machine. The bus-numbering scheme is bus-specific. Bus numbers may be virtual, but must match any numbering used by legacy interfaces such as **IoReport-ResourceUsage**.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Call **IoGetDeviceProperty** to get information about the bus to which a device is attached.

See Also

IoGetDeviceProperty

IRP_MN_QUERY_CAPABILITIES

The PnP Manager sends this IRP to get the capabilities of a device, such as whether the device can be locked or ejected.

Function and filter drivers can handle this request if they alter the capabilities supported by the bus driver. Bus drivers must handle this request for their child devices.

When Sent

The PnP Manager sends this IRP to the bus driver for a device immediately after the device is enumerated. The PnP Manager sends this IRP again after all the drivers for a device have started the device. A driver can send this IRP to get the capabilities for a device.

The PnP Manager and drivers send this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

Parameters.DeviceCapabilities.Capabilities points to a DEVICE_CAPABILITIES structure containing information about the capabilities of the device.

Output

Parameters.DeviceCapabilities.Capabilities points to the DEVICE_CAPABILITIES structure that reflects any modifications made by the driver(s) that handle the IRP.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_UNSUCCESSFUL.

If a function or filter driver does not handle this IRP, it calls **IoSkipCurrentIrpStack-Location** and passes the IRP down to the next driver. Such a driver must not modify **Irp->IoStatus.Status** and must not complete the IRP.

A bus driver sets **Irp->IoStatus.Status** and completes the IRP.

Operation

When a device is enumerated, but before the function and filter drivers are loaded for the device, the PnP Manager sends an IRP_MN_QUERY_CAPABILITIES request to the parent bus driver for the device. The bus driver must set any relevant values in the DEVICE_CAPABILITIES structure and return it to the PnP Manager.

After the device stack is built and drivers have started the device, the PnP Manager sends this IRP again to be handled first by the driver at the top of the device stack and then by each lower driver in the stack. Function and filter drivers can set an IoCompletion routine and handle this IRP on its way back up the device stack.

Drivers should add capabilities before they pass the IRP to the next lower driver.

Drivers should remove capabilities after all lower drivers have finished with the IRP. A driver does not typically remove capabilities that have been set by other drivers, but it might do so if it has special information about the capabilities of the device in a certain configuration. See the *Plug and Play, Power Management, and Setup Design Guide* for information on postponing IRP processing until lower drivers have finished.

After a device is enumerated and its drivers are loaded, its capabilities should not change. A device's capabilities might change if the device is removed and re-enumerated.

When handling an IRP_MN_QUERY_CAPABILITIES IRP, the driver that is the power policy manager for the device should set an IoCompletion routine and copy the device power capabilities, such as the S-to-D power state mappings, on the IRP's way back up the device stack. To determine the power capabilities of a child device, the parent bus driver creates another query-capabilities IRP and sends the IRP to its parent driver. See *Reporting Device Power Capabilities* in Part 3, "Power Management," in the *Plug and Play, Power Management, and Setup Design Guide* for more information.

If a driver handles this IRP, it should check the DEVICE_CAPABILITIES.**Version** value. If that value is not a version that the driver supports, the driver should fail the IRP. If the version is supported, the driver should check the **Size** field. A driver should set only those fields that are within the bounds of the capabilities structure that it received as input.

Drivers that handle this IRP can set some DEVICE_CAPABILITIES fields but must not set the **Size** and **Version** fields. These fields are only set by the component that sent the IRP.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

A bus driver sends this IRP to the parent device stack when it handles an IRP_MN_QUERY_CAPABILITIES request for one of its child devices. Also, a driver might send this IRP to get the device capabilities for one of its devices. A single driver in the stack has only part of the capabilities information for the device; sending an IRP to the device stack enables it to gather the full picture, including modifications by any filter drivers, and so forth.

See the *Kernel-Mode Drivers Design Guide* for information on sending IRPs. The following steps apply specifically to this IRP:

- Allocate a DEVICE_CAPABILITIES structure from paged pool and initialize it to zeros. Initialize the **Size** to sizeof(DEVICE_CAPABILITIES), the **Version** to 1, and **Address** and **UINumber** to -1.
- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to IRP_MJ_PNP, set **MinorFunction** to IRP_MN_QUERY_CAPABILITIES, and set **Parameters.DeviceCapabilities** to a pointer to the allocated DEVICE_CAPABILITIES structure.
- Initialize **IoStatus.Status** to STATUS_NOT_SUPPORTED.
- Deallocate the IRP and the DEVICE_CAPABILITIES structure when they are no longer needed.

See Also

DEVICE_CAPABILITIES

IRP_MN_QUERY_DEVICE_RELATIONS

Bus drivers must handle this request for **BusRelations** for their adapter or controller (bus FDO). Filter drivers might handle this request for **BusRelations**. Bus drivers must handle this request for **TargetDeviceRelation** for their child devices (child PDOs). Function and filter drivers might handle this request for **RemovalRelations**. Bus drivers might handle this request for **EjectionRelations** for their child devices (child PDOs).

When Sent

The PnP Manager sends this IRP to gather information about devices with a relationship to the specified device.

The PnP Manager queries a device's **BusRelations** (child devices) when the device is enumerated and at other times while the device is active, such as when a driver calls **IoInvalidateDeviceRelations** to indicate that a child device has arrived or departed.

The PnP Manager queries a device's **RemovalRelations** before it removes a device's drivers or ejects the device and it queries for **EjectionRelations** before it ejects a device.

The PnP Manager queries a device's **TargetDeviceRelation** when a driver or user-mode application registers for PnP notification of an **EventCategoryTargetDeviceChange** on the device. The PnP Manager queries for the device that is associated with a particular file object. This is the only PnP IRP that has a valid file object parameter. A driver can query a device stack for **TargetDeviceRelation**.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

Parameters.QueryDeviceRelations.Type specifies the type of relations being queried. Possible values include **BusRelations**, **EjectionRelations**, **RemovalRelations**, and **TargetDeviceRelation**. **PowerRelations** is not used.

Irp->FileObject points to a valid file object only if **Parameters.QueryDeviceRelations.Type** is **TargetDeviceRelation**.

Output

Returned in the I/O status block.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to a failure status such as STATUS_INSUFFICIENT_RESOURCES.

On success, a driver sets **Irp->IoStatus.Information** to a PDEVICE_RELATIONS pointer that points to the requested relations information. The DEVICE_RELATIONS structure is defined as follows:

```
typedef struct _DEVICE_RELATIONS {
    ULONG Count;
    PDEVICE_OBJECT Objects[1];  // variable length
} DEVICE_RELATIONS, *PDEVICE_RELATIONS;
```

Operation

If a driver returns relations in response to this IRP, it allocates a DEVICE_RELATIONS structure from paged memory containing a count and the appropriate number of device object pointers. The PnP Manager frees the structure when it is no longer needed. If a driver replaces a DEVICE_RELATIONS structure allocated by another driver, it must free the previous structure.

A driver must reference the PDO of any device that it reports in this IRP (**ObReference-Object**). The PnP Manager removes the reference when appropriate.

A function or filter driver should be prepared to handle this IRP for a device any time after its AddDevice routine has completed for the device. Bus drivers should be prepared to handle a query for **BusRelations** immediately after a device is enumerated.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

The following subsections describe the specific actions for handling the various queries.

BusRelations

When the PnP Manager queries for the bus relations (child devices) of an adapter or controller, the bus driver must return a list of pointers to the PDOs of any devices physically present on the bus. The bus driver reports all devices, regardless of whether they have been started. The bus driver might need to power up its bus device to determine which children are present.

The bus driver that responds to this IRP is the function driver for the bus adapter or controller, not the parent bus driver for the bus that the adapter or controller is connected to. Function drivers for non-bus devices do not handle this query. Such drivers just pass the IRP to the next lower driver. (See Figure 2.1.) Filter drivers typically do not handle this query.

In the example shown in Figure 2.1, the PnP Manager sends an IRP_MN_QUERY_ DEVICE_RELATIONS for **BusRelations** to the drivers for the USB hub device. The PnP Manager is requesting a list of the hub device's children.

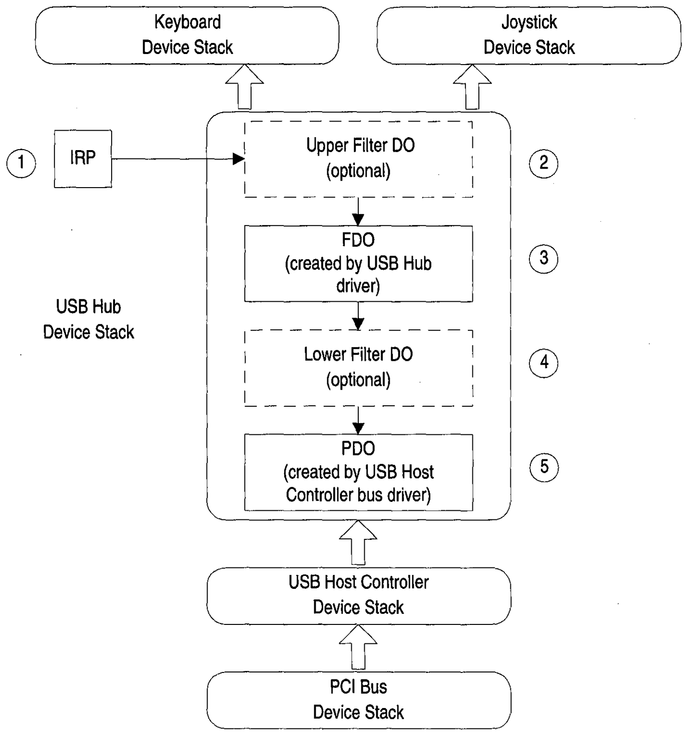

Keyboard Device Stack
Joystick Device Stack
1
IRP
Upper Filter DO (optional)
2
FDO (created by USB Hub driver)
3
USB Hub Device Stack
Lower Filter DO (optional)
4
PDO (created by USB Host Controller bus driver)
5
USB Host Controller Device Stack
PCI Bus Device Stack

Figure 2.1 Drivers Handling a Query For Bus Relations

1. As with all PnP IRPs, the PnP Manager sends the IRP to the top driver in the device stack for the device.
2. An optional filter driver might be the top driver in the stack. A filter driver typically does not handle this IRP; it passes the IRP down the stack. A filter driver might handle this IRP, for example, if the driver exposes a non-enumerable device on the bus.
3. The USB hub bus driver handles the IRP.

   The USB hub bus driver:

   - Creates a PDO for any child device that does not already have one.
   - Marks the PDO inactive for any device that is no longer present on the bus. The bus driver does not delete such PDOs. See *Removing a Device* in Part 2, "Plug and Play," in the *Plug and Play, Power Management, and Setup Design Guide* for information on when to delete the PDOs.
   - Reports any child devices that are present on the bus.

For each child device, the bus driver references the PDO and puts a pointer to the PDO in the DEVICE_RELATIONS structure.

There are two PDOs in this example, one for the joystick device and one for the keyboard device.

The bus driver should check whether another driver already created a DEVICE_ RELATIONS structure for this IRP. If so, the bus driver must add to the existing information.

If there is no child device present on the bus, the driver sets the count to zero in the DEVICE_RELATIONS structure and returns success.

- Sets the appropriate values in the I/O status block and passes the IRP to the next lower driver. The bus driver for the adapter or controller does not complete the IRP.

4. An optional lower filter, if present, typically does not handle this IRP. Such a filter driver passes the IRP down the stack. If a lower-filter driver handles this IRP, it can add PDO(s) to the list of child devices but it must not delete any PDOs created by other drivers.

5. The parent bus driver does not handle this IRP, unless it is the only driver in the device stack (the device is in raw mode). As with all PnP IRPs, the parent bus driver completes the IRP with **IoCompleteRequest**.

   If there are one or more bus filter drivers in the device stack, such drivers might handle the IRP on its way down to the bus driver and/or on the IRP's way back up the device stack (if there are IoCompletion routines). According to the PnP IRP rules, such a driver can add PDOs to the IRP on its way down the stack and/or modify the relations list on the IRP's way back up the stack (in IoCompletion routines).

EjectionRelations

A driver returns pointers to PDOs of any devices that might be physically removed from the system when the specified device is ejected. Do not report the PDOs of children of the device; the PnP Manager always requests that child devices be removed before their parent device.

The PnP Manager sends an IRP_MN_EJECT IRP to a device being ejected. The driver for such a device also receive a remove IRP. The device's ejection relations receive an IRP_ MN_REMOVE_DEVICE IRP (not an IRP_MN_EJECT IRP).

Only a parent bus driver can respond to an **EjectionRelations** query for one of its child devices. Function and filter drivers must pass it to the next lower driver in the device stack. If a bus driver receives this IRP as the function driver for its adapter or controller, the bus driver is performing the tasks of a function driver and must pass the IRP to the next lower driver.

PowerRelations

Reserved.

RemovalRelations

A driver returns pointers to PDOs of any devices whose drivers must be removed when the drivers for the specified device are removed. Do not report the PDOs of children of the device; the PnP Manager already requests removal of child devices before removing a device.

The order in which removal relations are removed is undefined. Removal relations at the same level in the device tree can be removed in any order.

Any driver in the device stack can handle this type of relations query. A function or filter driver handles the IRP before passing it to the next lower driver. A bus driver handles the IRP and then completes it.

TargetDeviceRelation

A parent bus driver must handle this type of relations query for its child devices. The bus driver references the child device's PDO with **ObReferenceObject** and returns a pointer to the PDO in the DEVICE_RELATIONS structure. There is only one PDO pointer in the structure for this relation type. The PnP Manager removes the reference to the PDO when the driver or application unregisters for notification on the device.

Only a parent bus driver responds to a **TargetDeviceRelation** query. Function and filter drivers must pass it to the next lower driver in the device stack. If a bus driver receives this IRP as the function driver for its adapter or controller, the bus driver is performing the tasks of a function driver and must pass the IRP to the next lower driver.

If a driver is not in a PDO-based stack, the driver sends a new target-device-relation query IRP to the device object associated with the file handle on which the driver performs I/O.

Sending This IRP

Drivers must not send this IRP to request **BusRelations**. Drivers are not restricted from sending this IRP for **RemovalRelations** or **EjectionRelations**, but it is not likely that a driver would do so.

Drivers can query a device stack for **TargetDeviceRelation**. See the *Kernel-Mode Drivers Design Guide* for information on sending IRPs. The following steps apply specifically to this IRP:

- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to IRP_MJ_PNP, set **MinorFunction** to IRP_MN_QUERY_DEVICE_RELATIONS, set **Parameters.QueryDeviceRelations.Type** to **TargetDeviceRelation**, and set **Irp->FileObject** to a valid file object.

- Initialize **IoStatus.Status** to STATUS_NOT_SUPPORTED.

If a driver sent this IRP to get the PDO to report in response to an IRP_MN_QUERY_DEVICE_RELATIONS for **TargetDeviceRelation** that *the driver* received, then the driver reports the PDO and frees the returned relations structure when the IRP completes. If a driver initiated this IRP for another reason, the driver frees the relations structure when the IRP completes and dereferences the PDO when it is no longer needed.

See Also

IoInvalidateDeviceRelations, IRP_MN_EJECT, IRP_MN_REMOVE_DEVICE, **IoRegisterPlugPlayNotification**, **ObReferenceObject**

IRP_MN_QUERY_DEVICE_TEXT

The PnP Manager uses this IRP to get a device's description or location information.

Bus drivers must handle this request for their child devices if the bus supports this information. Function and filter drivers do not handle this IRP.

When Sent

The PnP Manager sends two of these IRPs when a device is enumerated: one to query the device description and one to query the location information.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

Parameters.QueryDeviceText.DeviceTextType is a DEVICE_TEXT_TYPE specifying which string is requested. Possible values for DEVICE_TEXT_TYPE include **DeviceTextDescription** and **DeviceTextLocationInformation**.

Parameters.QueryDeviceText.LocaleId is an LCID specifying the locale for the requested text.

Output

Returned in the I/O status block.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a bus driver sets **Irp->IoStatus.Information** to a pointer to a driver-allocated block of memory containing a WCHAR buffer with the requested information. On an error, the bus driver sets **Irp->IoStatus.Information** to zero.

Operation

Bus drivers are strongly encouraged to return device descriptions for their child devices. This string is displayed in the "found new hardware" pop-up window if no INF match is found for the device.

Bus drivers are also encouraged to return **LocationInformation** for their child devices, but this information is optional. The format of this string depends on the bus. The Device Manager displays this string in the general properties tab for the device. Vendors should choose a string that conveys useful information to users and support personnel. For example, for PCI, the string contains the bus, device, and function. For PC Card, the string contains the slot.

If a bus driver returns information in response to this IRP, it allocates a NULL-terminated Unicode string from paged memory. The PnP Manager frees the string when it is no longer needed.

If a device does not provide description or location information, the device's parent bus driver completes the IRP (**IoCompleteRequest**) without modifying **Irp->IoStatus.Status** or **Irp->IoStatus.Information**.

Function and filter drivers do not handle this IRP; they pass it to the next lower driver with no changes to **Irp->IoStatus**.

Drivers for busses that support different text strings for different locales should be able to handle a request for a language that is not explicitly supported by the device. In such a situation, the bus driver should return the closest match for the locale or should fallback and return some appropriate supported locale string.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

IRP_MN_QUERY_ID

Bus drivers must handle requests for **BusQueryDeviceID** for their child devices (child PDOs). Bus drivers can handle requests for **BusQueryHardwareIDs**, **BusQuery-CompatibleIDs**, and **BusQueryInstanceID** for their child devices. Function and filter drivers do not handle this IRP.

When Sent

The PnP Manager sends this IRP when a device is enumerated. A driver might send this IRP to retrieve the instance ID for one of its devices.

The PnP Manager and drivers send this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

Parameters.QueryId.IdType specifies the kind of ID(s) requested. Possible values include **BusQueryDeviceID**, **BusQueryHardwareIDs**, **BusQueryComptibleIDs**, and **BusQuery-InstanceID**. The following ID type is reserved: **BusQueryDeviceSerialNumber**.

Output

Returned in the I/O status block.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a driver sets **Irp->IoStatus.Information** to a WCHAR pointer that points to the requested information. On error, a driver sets **Irp->IoStatus.Information** to zero.

Operation

If a driver returns ID(s) in response to this IRP, it allocates a WCHAR structure from paged pool to contain the ID(s). The PnP Manager frees the structure when it is no longer needed.

A driver returns a single string in response to a **BusQueryDeviceID** or a **BusQuery-InstanceID** request, and a MULTI_SZ string in response to a **BusQueryHardwareIDs** or a **BusQueryComptibleIDs** request.

If a driver returns an ID with an illegal character, the system will bugcheck. Characters with the following values are illegal in an ID for this IRP:

Less than 0x20 (' ')
Greater than 0x7F
Equal to 0x2C (',')

A driver must conform to the following length restrictions for IDs:

- Each hardware ID or compatible ID that a driver returns in this IRP must be less than MAX_DEVICE_ID_LEN characters long. This constant currently has a value of 200 as defined in *sdk\inc\cfgmgr32.h.*
- If a bus driver supplies globally unique instance IDs for its child devices (that is, the driver sets DEVICE_CAPABILITIES.**UniqueID** for the devices), then the combination of device ID plus instance ID must be less than (MAX_DEVICE_ID_LEN - 1) characters. The OS requires the additional character for a path separator.

- If a bus driver does not supply globally unique instance IDs for its child devices, then the combination of device ID plus instance ID must be less than (MAX_DEVICE_ID_LEN - 28). The value of this equation is currently 172.

Bus drivers should be prepared to handle this IRP for a child device immediately after the device is enumerated.

Specifying BusQueryDeviceID and BusQueryInstanceID

The values a bus driver supplies for **BusQueryDeviceID** and **BusQueryInstanceID** allow the OS to differentiate a device from other devices on the machine. The OS uses the device ID, instance ID, and the unique ID field returned in the IRP_MN_QUERY_DEVICE_CAPABILITIES IRP to locate registry information for device.

For **BusQueryDeviceID**, a bus driver supplies the device's device ID. A device ID should contain the most-specific description of the device possible, incorporating the name of the enumerator and strings identifying the manufacturer, device, revision, packager, and packaged product, where possible. For example, the PCI bus driver responds with device IDs of the form PCI\VEN_xxxx&DEV_xxxx&SUBSYS_xxxxxxxx&REV_xx, encoding all five of the items mentioned above. However, a device ID should *not* contain enough information to differentiate between two identical devices. This information should be encoded in the instance ID.

For **BusQueryInstanceID**, a bus driver should supply a string that contains the instance ID for the device. Setup and bus drivers use the instance ID, with other information, to differentiate between two identical devices on the machine. The instance ID is either unique across the whole machine or just unique on the device's parent bus.

If an instance ID is only unique on the bus, the bus driver specifies that string for **BusQueryInstanceID** but also specifies a **UniqueID** value of FALSE in response to an IRP_MN_QUERY_CAPABILITIES request for the device. If **UniqueID** is FALSE, the PnP Manager enhances the instance ID by adding information about the device's parent and thus makes the ID unique on the machine. In this case the bus driver should not take extra steps to make its devices' instance IDs globally unique; just return the appropriate capabilities information and the OS takes care of it.

If a bus driver can supply a globally unique ID for each child device, such as a serial number, the bus driver specifies those strings for **BusQueryInstanceID** and specifies a **UniqueID** value of TRUE in response to an IRP_MN_QUERY_CAPABILITIES request for each device.

Specifying BusQueryHardwareIDs and BusQueryCompatibleIDs

The values a bus driver supplies for **BusQueryHardwareIDs** and **BusQueryCompatibleIDs** allow Setup to locate the appropriate drivers for the bus's child device.

A bus driver responds to each of these requests with a list of IDs that describe the device. When returning more than one hardware ID and/or more than one compatible ID, a bus driver should list the IDs in the order of most-specific to most-general to facilitate choosing the best driver match for the device. The first entry in the hardware IDs list is the most-specific description of the device and, as such, it is usually identical to the device ID.

Setup checks the IDs against the IDs listed in INF files for possible matches. Setup first scans the hardware IDs list, then the compatible IDs list. Earlier entries are treated as more specific descriptions of the device, and later entries as more general (and thus less optimal) matches for the device. If no match is found in the list of hardware IDs, Setup might prompt the user for installation media before moving on to the list of compatible IDs.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Typically, only the PnP Manager sends this IRP.

To get the hardware IDs or compatible IDs for a device, call **IoGetDeviceProperty** instead of sending this IRP.

A driver might send this IRP to retrieve the instance ID for one of its devices. For example, consider a multifunction PnP ISA device whose functions do not operate independently. The PnP Manager enumerates the functions as separate devices, but the driver for such a device might be required to associate one or more of the functions. Because PnP ISA guarantees a unique instance ID, the driver for such a multifunction device can use the instance IDs to locate functions that reside on the same device. The driver for such a device must also get the device's enumerator name by calling **IoGetDeviceProperty**, to confirm that the device is a PnP ISA device.

See the *Kernel-Mode Drivers Design Guide* for information on sending IRPs. The following steps apply specifically to this IRP:

- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to IRP_MJ_PNP, set **MinorFunction** to IRP_MN_QUERY_ID, and set **Parameters.QueryId.IdType** to **BusQueryInstanceID**.

- Set **IoStatus.Status** to STATUS_NOT_SUPPORTED.

In addition to sending the query ID IRP, the driver must call **IoGetDeviceProperty** to get the **DevicePropertyEnumeratorName** for the device.

After the IRP completes and the driver is finished with the ID, the driver must free the ID structure returned by the driver(s) that handled the query IRP.

See Also

IoGetDeviceProperty

IRP_MN_QUERY_INTERFACE

The IRP_MN_QUERY_INTERFACE request enables a driver to export a direct-call interface to other drivers.

A bus driver that exports an interface must handle this request for its child devices (child PDOs). Function and filter can optionally handle this request.

An "interface" in this context consists of routine(s) and possibly data exported by a driver or set of drivers. An interface has a structure that describes its contents and a GUID that identifies its type.

For example, the PCMCIA bus driver exports an interface of type GUID_PCMCIA_INTERFACE_STANDARD that contains routines for operations such as getting the write-protect condition of a PCMCIA memory card. The function driver for such a memory card can send an IRP_MN_QUERY_INTERFACE request to the parent PCMCIA bus driver to get pointers to the PCMCIA interface routines.

This section describes the query-interface IRP as a general mechanism. Drivers that expose an interface should provide additional information about their specific interface.

When Sent

A driver or system component sends this IRP to get information about an interface exported by a driver for a device.

A driver or system component sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

A driver can receive this IRP at any time after the driver's AddDevice routine has been called for the device. The device may or may not be started when this IRP is sent (the driver cannot assume that it has successfully completed an IRP_MN_START_DEVICE request for the device).

Input

Parameters.QueryInterface is a structure that describes the interface being requested. The structure contains the following information:

```
CONST GUID *InterfaceType;
USHORT Size;
USHORT Version;
PINTERFACE Interface;
PVOID InterfaceSpecificData
```

The members of the structure are defined as follows:

InterfaceType

Points to a GUID that identifies the interface being requested. The GUID can be for a system-defined interface, such as GUID_BUS_INTERFACE_STANDARD, or a custom interface. The GUIDs for system-defined interfaces are listed in *wdmguid.h.* GUIDs for custom interfaces should be generated with **uuidgen**.

Size

Specifies the size of the interface being requested. Drivers that handle this IRP must not return an INTERFACE structure larger than **Size** bytes.

Version

Specifies the version of the interface being requested.

If a driver supports more than one version of an interface, the driver returns the closest supported version without exceeding the requested version. The component that sent the IRP should examine the returned **Interface.Version** field and determine what to do based on that value.

Interface

Points to an INTERFACE structure in which to return the requested interface information. The component sending the IRP allocates this structure from paged memory.

The base layout for an INTERFACE structure is defined as follows:

```
typedef VOID (*PINTERFACE_REFERENCE)(PVOID Context);
typedef VOID (*PINTERFACE_DEREFERENCE)(PVOID Context);

typedef STRUCT _INTERFACE {
    USHORT Size;
    USHORT Version;
    PVOID Context;
    PINTERFACE_REFERENCE InterfaceReference;
    PINTERFACE_DEREFERENCE InterfaceDereference;
    // interface-specific entries go here
} INTERFACE, *PINTERFACE;
```

A driver that exports an interface defines a new type containing the members shown above plus members for routines and/or data in the interface. (The driver also defines a GUID for the interface, as described in the **InterfaceType** member, above.)

A driver that exports an interface defines the execution environment for each routine in the interface, including the IRQL at which the routine can be called, and so forth.

InterfaceSpecificData

Specifies additional information about the interface being requested.

For some interfaces, the component sending the IRP specifies additional information in this field. Typically, this field is NULL and the **InterfaceType** and **Version** are sufficient to identify the interface being requested.

Output

On success, a driver fills in the members of the **Parameters.QueryInterface.Interface** structure.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a bus driver sets **Irp->IoStatus.Information** to zero.

If a function or filter driver does not handle this IRP, it calls **IoSkipCurrentIrpStack-Location** and passes the IRP down to the next driver. Such a driver must not modify **Irp->IoStatus.Status** and must not complete the IRP.

If a bus driver does not export the requested interface and therefore does not handle this IRP for a child PDO, the bus driver leaves **Irp->IoStatus.Status** as is and completes the IRP.

Operation

A driver handles this IRP if the parameters specify an interface the driver supports.

A driver must not queue this IRP if the IRP requests an interface that the driver does not support. A driver must check **Parameters.QueryInterface.InterfaceType**. If the interface is not one the driver supports, the driver must pass the IRP to the next lower driver in the device stack without blocking.

A driver that returns an interface in response to this IRP must reference the interface. The component that requested the interface by sending the IRP is responsible for dereferencing the interface (using the interface's **InterfaceDereference** routine). If the component that sends the IRP, driver X, passes the interface to another component, driver Y, then driver X is responsible for taking out another reference on the interface (**InterfaceReference**) and driver Y is responsible for removing the additional reference (**InterfaceDereference**).

A driver that handles this IRP should avoid passing the IRP to another device stack to get the requested interface. Such a design would create dependencies between the device stacks that are difficult to manage. For example, the device represented by the second device stack cannot be removed until the appropriate driver in the first stack dereferences the interface.

Interfaces can be bus-specific or bus-independent. Bus-specific interfaces are defined in the header files for those buses. The system defines a bus-independent interface, BUS_INTERFACE_STANDARD, for exporting standard bus interfaces. This interface has the type GUID_BUS_INTERFACE_STANDARD and is defined in *wdm.h* as follows:

```
typedef BOOLEAN (*PTRANSLATE_BUS_ADDRESS)(
    IN PVOID Context,
    IN PHYSICAL_ADDRESS BusAddress,
    IN ULONG Length,
    IN OUT PULONG AddressSpace,
    OUT PPHYSICAL_ADDRESS TranslatedAddress
    );

typedef struct _DMA_ADAPTER *(*PGET_DMA_ADAPTER)(
    IN PVOID Context,
    IN struct _DEVICE_DESCRIPTION *DeviceDescriptor,
    OUT PULONG NumberOfMapRegisters
    );

typedef ULONG (*PGET_SET_DEVICE_DATA)(
    IN PVOID Context,
    IN ULONG DataType,
    IN PVOID Buffer,
    IN ULONG Offset,
    IN ULONG Length
    );

typedef struct _BUS_INTERFACE_STANDARD {
    //
    // generic interface header
    //
    USHORT Size;
    USHORT Version;
    PVOID Context;
    PINTERFACE_REFERENCE InterfaceReference;
    PINTERFACE_DEREFERENCE InterfaceDereference;
    //
    // standard bus interfaces
    //
    PTRANSLATE_BUS_ADDRESS TranslateBusAddress;
    PGET_DMA_ADAPTER GetDmaAdapter;
    PGET_SET_DEVICE_DATA SetBusData;
    PGET_SET_DEVICE_DATA GetBusData;

} BUS_INTERFACE_STANDARD, *PBUS_INTERFACE_STANDARD;
```

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

This IRP is used specifically to communicate routine entry points between layered drivers for a device. A *device interface* is a separate mechanism, primarily for exposing a path to a device for use by user-mode components or other kernel components. Call **IoGetDevice-Interfaces** to get a list of device interfaces of a particular device interface class, such as all the devices on the system that support a HID interface.

Sending This IRP

See the *Kernel-Mode Drivers Design Guide* for information on sending IRPs. The following steps apply specifically to this IRP:

- Allocate an INTERFACE structure from paged pool and initialize it to zeros.
- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to IRP_MJ_PNP, set **MinorFunction** to IRP_MN_QUERY_INTERFACE, and set the appropriate values in **Parameters.QueryInterface**.
- Initialize **IoStatus.Status** to STATUS_NOT_SUPPORTED.
- Deallocate the IRP and the INTERFACE structure when they are no longer needed.
- Use the interface routines and context parameter as described in the specification for the interface.
- Decrement the reference count using the **InterfaceDereference** routine when the interface is no longer needed. Do not call any interface routines after dereferencing the interface.

A driver typically sends this IRP to the top of the device stack in which the driver is attached. If a driver sends this IRP to a different device stack, the driver must register for target device notification on the other device if the other device is not an ancestor of the device that the driver is servicing. Such a driver calls **IoRegisterPlugPlayNotification** with an *EventCategory* of **EventCategoryTargetDeviceChange**. When the driver receives notification of type GUID_TARGET_DEVICE_QUERY_REMOVE, the driver must de-reference the interface. The driver can requery for the interface if it receives a subsequent GUID_TARGET_DEVICE_REMOVE_CANCELLED notification.

See Also

IoGetDeviceInterfaces, **IoRegisterPlugPlayNotification**

IRP_MN_QUERY_LEGACY_BUS_INFORMATION

This IRP is reserved for system use.

IRP_MN_QUERY_PNP_DEVICE_STATE

Function, filter, and bus drivers can handle this request.

When Sent

The PnP Manager sends this IRP after the drivers for a device return success from the IRP_MN_START_DEVICE request sent when a device is first started. This IRP is not sent on a start after a stop for resource rebalancing. The PnP Manager also sends this IRP when a driver for the device calls **IoInvalidateDeviceState**.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in the context of an arbitrary thread.

Input

None

Output

Returned in I/O status block.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_UNSUCCESSFUL.

On success, a driver sets **Irp->IoStatus.Information** to a PNP_DEVICE_STATE bitmask.

If a function or filter driver does not handle this IRP, it calls **IoSkipCurrentIrpStack-Location**, does not set an IoCompletion routine, and passes the IRP down to the next driver. Such a driver must not modify **Irp->IoStatus** and must not complete the IRP.

If a bus driver does not handle this IRP, it leaves **Irp->IoStatus.Status** as is and completes the IRP.

Operation

This IRP is handled first by the driver at the top of the device stack and then by each next lower driver in the stack.

A driver handles this IRP if it has information about the PnP state of a device. A driver can set or clear the flags in the PNP_DEVICE_STATE bitmask. If another driver has set a PNP_DEVICE_STATE in **Irp->IoStatus.Information**, a driver must take care to modify the flags in that bitmask rather than overwrite the whole structure.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

See Also

IoInvalidateDeviceState, PNP_DEVICE_STATE

IRP_MN_QUERY_REMOVE_DEVICE

All PnP drivers must handle this IRP.

When Sent

The PnP Manager sends this IRP to inform drivers that a device is about to be removed from the machine and to query whether the device can be removed without disrupting the machine. The PnP Manager also sends this IRP if a user requests to update driver(s) for the device.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input

None

Output

None

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_UNSUCCESSFUL.

Operation

This IRP is handled first by the driver at the top of the device stack and then passed down to each lower driver in the stack.

In response to this IRP, drivers indicate whether the device can be removed without disrupting the machine.

See *Removing a Device* in Part 2, "Plug and Play," in the *Plug and Play, Power Management, and Setup Design Guide* for detailed information on handling remove IRPs. Also see that *Design Guide* for the general rules for handling all PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

See Also

IRP_MN_CANCEL_REMOVE_DEVICE, IRP_MN_DEVICE_USAGE_NOTIFICATION, IRP_MN_REMOVE_DEVICE

IRP_MN_QUERY_RESOURCE_REQUIREMENTS

The PnP Manager uses this IRP to get a device's resource requirements list.

Bus drivers must handle this request for their child devices that require hardware resources. Bus filter drivers can handle this request. Function and filter drivers do not handle this IRP.

When Sent

The PnP Manager sends this IRP when a device is enumerated, prior to allocating resources to a device, and when a driver reports that its device's resource requirements have changed.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

None

Output

Returned in the I/O status block.

I/O Status Block

A driver that handles this IRP sets **Irp->IoStatus.Status** to STATUS_SUCCESS or an appropriate error status.

On success, a driver sets **Irp->IoStatus.Information** to a pointer to an IO_RESOURCE_REQUIREMENTS_LIST that contains the requested information. On an error, the driver sets **Irp->IoStatus.Information** to zero.

Operation

If a bus driver returns a resource requirements list in response to this IRP, it allocates an IO_RESOURCE_REQUIREMENTS_LIST from paged memory. The PnP Manager frees the buffer when it is no longer needed.

If a device requires no hardware resources, the device's bus driver completes the IRP (**IoCompleteRequest**) without modifying **Irp->IoStatus.Status** or **Irp->IoStatus.Information**.

If a bus filter driver handles this IRP, it modifies the resource requirements list created by the bus driver. A bus filter driver modifies the list on the IRP's way back up the device stack. A bus filter driver must preserve the order of resources in the resource requirements list and must not alter resource tags that it does not handle. If a bus filter driver changes the size of the resource requirements list, the driver must allocate a new structure from paged memory and free the previous structure. If a bus filter driver adds a new resource requirement to the list and the resource is assigned to the device, the driver must filter the new resource out of the IRP_MN_START_DEVICE IRP so it is not passed to the bus driver.

Function and non-bus filter drivers do not handle this IRP; they pass it to the next lower driver with no changes to **Irp->IoStatus**.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

See Also

IO_RESOURCE_REQUIREMENTS_LIST

IRP_MN_QUERY_RESOURCES

The PnP Manager uses this IRP to get a device's boot configuration resources.

Bus drivers must handle this request for their child devices that require hardware resources. Function and filter drivers do not handle this IRP.

When Sent

The PnP Manager sends this IRP when a device is enumerated.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

None

Output

Returned in the I/O status block.

I/O Status Block

A bus driver that handles this IRP sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a bus driver sets **Irp->IoStatus.Information** to a pointer to a CM_RESOURCE_LIST that contains the requested information. On an error, the bus driver sets **Irp->IoStatus.Information** to zero.

Operation

If a bus driver returns a resource list in response to this IRP, it allocates a CM_RESOURCE_LIST from paged memory. The PnP Manager frees the buffer when it is no longer needed.

If a device requires no hardware resources, the device's parent bus driver completes the IRP (**IoCompleteRequest**) without modifying **Irp->IoStatus.Status** or **Irp->IoStatus.Information**.

Function and filter drivers do not receive this IRP.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

Drivers can call **IoGetDeviceProperty** to get the boot configuration for a device, in both raw and translated forms.

See Also

CM_RESOURCE_LIST, **IoGetDeviceProperty**

IRP_MN_QUERY_STOP_DEVICE

All PnP drivers must handle this IRP.

When Sent

The PnP Manager sends this IRP to query whether a device can be stopped for resource rebalancing.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input

None

Output

None

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status. If a driver cannot stop the device, the driver sets **Irp->IoStatus.Status** to STATUS_UNSUCCESSFUL.

A bus driver can set **Irp->IoStatus.Status** to STATUS_RESOURCE_REQUIREMENTS_CHANGED to indicate success for the IRP but also to request that the PnP Manager requery the resource requirements for the device before sending the stop IRP.

Operation

This IRP is handled first by the driver at the top of the device stack and then passed down to each lower driver in the stack.

In response to this IRP, drivers indicate whether it is safe to stop the device for resource rebalancing.

See *Stopping a Device for Resource Rebalancing* in Part 2, "Plug and Play," in the *Plug and Play, Power Management, and Setup Design Guide* for detailed information on handling stop IRPs. Also see that *Design Guide* for the general rules for handling all PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

See Also

IRP_MN_CANCEL_STOP_DEVICE, IRP_MN_DEVICE_USAGE_NOTIFICATION, IRP_MN_START_DEVICE, IRP_MN_STOP_DEVICE

IRP_MN_READ_CONFIG

Bus drivers for buses with configuration space must handle this request for their child devices (child PDOs). Filter and function drivers do not handle this request.

When Sent

A driver or other system component sends this IRP to read the configuration space of a device's parent bus.

A driver or other system component sends this IRP at IRQL < DISPATCH_LEVEL in an arbitrary thread context.

Input

Parameters.ReadWriteConfig is a structure containing the following information:

```
ULONG WhichSpace;
PVOID Buffer;
ULONG Offset;
ULONG Length
```

The members of the structure can be interpreted differently by different bus drivers, but the members are typically defined as follows:

WhichSpace

Specifies the configuration space.

Buffer

Points to a buffer in which to return the requested information. The component sending the IRP allocates this structure from paged memory. The format of the buffer is bus-specific.

Offset

Specifies an offset into the configuration space.

Length

Specifies the number of bytes to read.

Output

On success, a bus driver fills the buffer at **Parameters.ReadWriteConfig.Buffer** with the requested data.

I/O Status Block

A bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_INVALID_PARAMETER_*n*, STATUS_NO_SUCH_DEVICE, or STATUS_DEVICE_NOT_READY.

On success, a bus driver sets **Irp->IoStatus.Information** to the number of bytes returned.

If a bus driver is unable to complete this request immediately it can mark the IRP pending, return STATUS_PENDING, and complete the IRP at a later time.

Operation

A bus driver handles this IRP for its child devices (child PDOs).

Function and filter drivers do not handle this IRP; they pass it to the next lower driver with no changes to **Irp->IoStatus.Status** and do not set an IoCompletion routine.

A bus driver that handles this request should check the *WhichSpace* parameter to ensure that it contains a value that the driver supports.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Typically, a function driver sends this IRP to the top driver in the device stack to which it is attached and the IRP is handled by the parent bus driver.

See the *Kernel-Mode Drivers Design Guide* for information on sending IRPs. The following steps apply specifically to this IRP:

- Allocate a buffer from paged pool and initialize it to zeros.
- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to IRP_MJ_PNP, set **MinorFunction** to IRP_MN_READ_CONFIG, and set the appropriate values in **Parameters.ReadWriteConfig**.
- Initialize **IoStatus.Status** to STATUS_NOT_SUPPORTED.
- Deallocate the IRP and the buffer when they are no longer needed.

Drivers must send this IRP from IRQL < DISPATCH_LEVEL.

A driver can access a bus's configuration space at DISPATCH_LEVEL through a bus interface routine, if the parent bus driver supports such an interface. To get a bus interface, a driver sends an IRP_MN_QUERY_INTERFACE request to the device stack in which the driver is attached. The driver then calls the appropriate routine returned in the interface.

For example, to read configuration space from DISPATCH_LEVEL a driver can call IRP_MN_QUERY_INTERFACE during driver initialization to get the BUS_INTERFACE_STANDARD interface from the parent bus driver. The driver sends the query IRP from IRQL PASSIVE_LEVEL. Later, from code at IRQL DISPATCH_LEVEL, the driver calls the appropriate routine returned in the interface, such as the **Interface.GetBusData** routine.

See Also

IRP_MN_QUERY_INTERFACE, IRP_MN_WRITE_CONFIG

IRP_MN_REMOVE_DEVICE

All PnP drivers must handle this IRP.

When Sent

The PnP Manager uses this IRP to direct drivers to remove a device's software representation (device objects, and so forth). The PnP Manager sends this IRP when a device has been removed in an orderly fashion (for example, initiated by a user in the Unplug or Eject Hardware applet), by surprise (a user pulls the device from its slot without prior warning), or when the user requests to update driver(s). The PnP Manager also sends this IRP if one of the drivers in the device stack fails an IRP_MN_START_DEVICE request for the device.

For an orderly device removal, the PnP Manager sends an IRP_MN_QUERY_REMOVE_DEVICE prior to the remove IRP. In this case, the device is in the remove-pending state when the remove IRP arrives. For a surprise device removal on Microsoft Windows 2000, the PnP Manager sends an IRP_MN_SURPRISE_REMOVAL prior to the remove IRP. In this case, the device is in the surprise-removed state when the remove IRP arrives. Drivers can also receive a remove IRP before a device is started. In this case, the device is in the non-started state when the IRP arrives.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input

None

Output

None

I/O Status Block

A driver must set **Irp->IoStatus.Status** to STATUS_SUCCESS. Drivers must not fail this IRP.

Operation

This IRP is handled first by the driver at the top of the device stack and then by each lower driver in the stack.

In response to this IRP, drivers perform such tasks as powering down the device, removing the device's software representation (device objects, and so forth), and releasing any resources for the device. See *Removing a Device* in Part 2, "Plug and Play," in the *Plug and Play, Power Management, and Setup Design Guide* for detailed information on handling remove IRPs in function, filter, and bus drivers.

Also see the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling all PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

If a bus driver detects that one (or more) of its child devices (child PDOs) has been physically removed from the machine, the bus driver calls **IoInvalidateDeviceRelations** to report the change to the PnP Manager. The PnP Manager then sends remove IRPs for any devices that have disappeared.

See Also

IoInvalidateDeviceRelations, **IoRegisterPlugPlayNotification**, IRP_MN_CANCEL_REMOVE_DEVICE, IRP_MN_QUERY_REMOVE_DEVICE, IRP_MN_SURPRISE_REMOVAL

IRP_MN_SET_LOCK

Bus drivers must handle this IRP for their child devices (child PDOs) that support device locking. Function and filter drivers do not handle this request.

When Sent

The PnP Manager sends this IRP to direct driver(s) to lock the device and prevent device eject, or to unlock the device.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

Parameters.SetLock.Lock is a BOOLEAN specifying whether to lock (TRUE) or unlock (FALSE) the device.

Output

None

I/O Status Block

A bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status.

On success, a driver sets **Irp->IoStatus.Information** to zero.

If a bus driver does not handle this IRP, it leaves **Irp->IoStatus.Status** as is and completes the IRP.

Function and filter drivers do not handle this IRP. Such drivers call **IoSkipCurrentIrp-StackLocation** and pass the IRP down to the next driver. Function and filter drivers do not set an IoCompletion routine, do not modify **Irp->IoStatus**, and must not complete the IRP.

Operation

If a driver returns success for this IRP, it ensures that the device has been locked or unlocked before completing the IRP.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

IRP_MN_START_DEVICE

All PnP drivers must handle this IRP.

When Sent

The PnP Manager sends this IRP after it has assigned hardware resources, if any, to the device. The device may have been recently enumerated and is being started for the first time, or the device may be restarting after being stopped for resource rebalancing.

Sometimes the PnP Manager sends an IRP_MN_START_DEVICE to a device that is already started, supplying a different set of resources than the device is currently using. A driver initiates this action by calling **IoInvalidateDeviceState** and responding to the subsequent IRP_MN_QUERY_PNP_DEVICE_STATE request with the PNP_RESOURCE_REQUIREMENTS_CHANGED flag set. A bus driver might use this mechanism, for example, to open a new aperture on a PCI-to-PCI bridge.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input

Parameters.StartDevice.AllocatedResources points to a CM_RESOURCE_LIST describing the hardware resources that the PnP Manager assigned to the device. This list contains the resources in raw form. Use the raw resources to program the device.

Parameters.StartDevice.AllocatedResourcesTranslated points to a CM_RESOURCE_LIST describing the hardware resources that the PnP Manager assigned to the device. This list contains the resources in translated form. Use the translated resources to connect the interrupt vector, map I/O space, and map memory.

Output

None

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_UNSUCCESSFUL or STATUS_INSUFFICIENT_RESOURCES.

If a driver requires some time to execute its start operations for a device, it can mark the IRP pending and return STATUS_PENDING.

Operation

This IRP must be handled first by the parent bus driver for a device and then by each higher driver in the device stack.

In response to this IRP, drivers start a device for the first time or restart a device that was stopped. The exact operations required to start a device vary from device to device, but can include powering on the device, performing device-specific initialization, and connecting the interrupt.

A driver can typically handle this IRP in the same way whether it is starting a device for the first time or restarting a device after an IRP_MN_STOP_DEVICE, except if a driver needs to restore device state on a restart after a stop.

See *Starting a Device* in Part 2, "Plug and Play," in the *Plug and Play, Power Management, and Setup Design Guide* for detailed information on handling a start IRP. Also see that *Design Guide* for the general rules for handling all PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

See Also

IRP_MN_STOP_DEVICE

IRP_MN_STOP_DEVICE

All PnP drivers must handle this IRP.

When Sent

The PnP Manager sends this IRP to stop a device so it can reconfigure the device's hardware resources. The PnP Manager sends this IRP only if a prior IRP_MN_QUERY_STOP_DEVICE completed successfully.

The PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input

None

Output

None

I/O Status Block

A driver must set **Irp->IoStatus.Status** to STATUS_SUCCESS.

Operation

This IRP is handled first by the driver at the top of the device stack and then passed down to each lower driver in the stack.

In response to this IRP, drivers stop the device and release any hardware resources being used by the device, such as I/O ports and interrupts.

On Windows 2000, a stop IRP is used solely to free a device's hardware resources so they can be reconfigured. Once the resources are reconfigured, the device is restarted. A stop IRP is not a precursor to a remove IRP. See the *Plug and Play, Power Management, and Setup Design Guide* for more information about the order in which PnP IRPs are sent to devices.

A driver must not fail this IRP. If a driver cannot release the device's hardware resources, it must fail the preceding query-stop IRP.

See *Stopping a Device for Resource Rebalancing* in Part 2, “Plug and Play,” in the *Plug and Play, Power Management, and Setup Design Guide* for detailed information on handling stop IRPs. Also see that *Design Guide* for the general rules for handling all PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

See Also

IRP_MN_QUERY_STOP_DEVICE, IRP_MN_START_DEVICE

IRP_MN_SURPRISE_REMOVAL

All PnP drivers must handle this IRP.

When Sent

The Windows 2000 PnP Manager sends this IRP to notify the drivers for a device that the device has been unexpectedly removed from the machine and is no longer available for I/O.

The Windows 2000 PnP Manager sends this IRP before notifying user-mode applications or other kernel-mode components. After this IRP completes, the PnP Manager notifies registered applications and drivers that the device has been removed.

The device can be in any PnP state when the PnP Manager sends this IRP.

The Windows 98 PnP Manager does not send this IRP.

The Windows 2000 PnP Manager sends this IRP at IRQL PASSIVE_LEVEL in the context of a system thread.

Input

None

Output

None

I/O Status Block

A driver must set **Irp->IoStatus.Status** to STATUS_SUCCESS. A driver must not fail this IRP.

Operation

This IRP is handled first by the driver at the top of the device stack and then passed down to each lower driver in the stack.

This IRP indicates that a user removed a hot-plug device, either on purpose or by accident, without first using the user interface that manages removal of the device, or that a driver for the device failed a start IRP after a successful stop IRP.

Because the device is no longer present on the machine, drivers must immediately stop all access to the device. A driver releases any resources associated with the device, but leaves its device object attached to the device stack until the PnP Manager sends a subsequent IRP_MN_REMOVE_DEVICE request. Drivers fail any outstanding I/O to the device.

See *Removing a Device* in Part 2, "Plug and Play," in the *Plug and Play, Power Management, and Setup Design Guide* for detailed information on handling this IRP and for the general rules for handling all PnP IRPs.

Sending This IRP

Reserved for system use. Drivers must not send this IRP.

See Also

IRP_MN_REMOVE_DEVICE

IRP_MN_WRITE_CONFIG

Bus drivers for buses with configuration space must handle this request for their child devices (child PDOs). Function and filter drivers do not handle this request.

When Sent

A driver or other system component sends this IRP to write data to the configuration space of a device's parent bus.

A driver or other system component sends this IRP at IRQL < DISPATCH_LEVEL in an arbitrary thread context.

Input

Parameters.ReadWriteConfig is a structure containing the following information:

```
ULONG WhichSpace;
PVOID Buffer;
ULONG Offset;
ULONG Length
```

The members of the structure can be interpreted differently by different bus drivers, but the members are typically defined as follows:

WhichSpace

Specifies the configuration space.

Buffer

Points to a buffer that contains the data to be written. The format of the buffer is bus-specific.

Offset

Specifies an offset into the configuration space.

Length

Specifies the number of bytes to be written.

Output

Returned in the I/O status block.

I/O Status Block

A bus driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS or to an appropriate error status such as STATUS_INVALID_PARAMETER_*n*, STATUS_NO_SUCH_DEVICE, or STATUS_DEVICE_NOT_READY.

On success, a bus driver sets **Irp->IoStatus.Information** to the number of bytes written.

If a bus driver is unable to complete this request immediately, it can mark the IRP pending, return STATUS_PENDING, and complete the IRP at a later time.

Operation

A bus driver handles this IRP for its child devices (child PDOs).

Function and filter drivers do not handle this IRP; they pass it to the next lower driver with no changes to **Irp->IoStatus.Status** and do not set an IoCompletion routine.

See the *Plug and Play, Power Management, and Setup Design Guide* for the general rules for handling PnP IRPs.

Sending This IRP

Typically, a function driver sends this IRP to the device stack to which it is attached and the IRP is handled by the parent bus driver.

See the *Kernel-Mode Drivers Design Guide* for information on sending IRPs. The following steps apply specifically to this IRP:

- Allocate a buffer from paged pool and initialize it with the data to be written.
- Set the values in the next I/O stack location of the IRP: set **MajorFunction** to IRP_MJ_PNP, set **MinorFunction** to IRP_MN_WRITE_CONFIG, and set the appropriate values in **Parameters.ReadWriteConfig**.
- Initialize **IoStatus.Status** to STATUS_NOT_SUPPORTED.
- Deallocate the IRP and the buffer when they are no longer needed.

Drivers must send this IRP from IRQL < DISPATCH_LEVEL.

A driver can access a bus's configuration space at DISPATCH_LEVEL through a bus interface routine, if the parent bus driver exports such an interface. To get a bus interface, a driver sends an IRP_MN_QUERY_INTERFACE request to its parent bus driver. The driver then calls the appropriate routine returned in the interface.

For example, to write configuration space from DISPATCH_LEVEL a driver can call IRP_MN_QUERY_INTERFACE during driver initialization to get the BUS_INTERFACE_STANDARD interface from the parent bus driver. The driver sends the query IRP from IRQL PASSIVE_LEVEL. Later, from code at IRQL DISPATCH_LEVEL, the driver calls the appropriate routine returned in the interface, such as the **Interface.SetBusData** routine.

See Also

IRP_MN_QUERY_INTERFACE, IRP_MN_READ_CONFIG

CHAPTER 3

Plug and Play Structures

This chapter describes the structures that are parameters to more than one PnP routine or IRP. Structures that are used by only one routine are described in the documentation for that routine or IRP.

See the *Plug and Play, Power Management, and Setup Design Guide* for background and task-oriented information on supporting PnP in drivers.

DEVICE_CAPABILITIES

```
typedef struct _DEVICE_CAPABILITIES {
  USHORT Size;
  USHORT Version;
  ULONG DeviceD1:1;
  ULONG DeviceD2:1;
  ULONG LockSupported:1;
  ULONG EjectSupported:1;
  ULONG Removable:1;
  ULONG DockDevice:1;
  ULONG UniqueID:1;
  ULONG SilentInstall:1;
  ULONG RawDeviceOK:1;
  ULONG SurpriseRemovalOK:1;
  ULONG WakeFromD0:1;
  ULONG WakeFromD1:1;
  ULONG WakeFromD2:1;
  ULONG WakeFromD3:1;
  ULONG HardwareDisabled:1;
  ULONG NonDynamic:1;
  ULONG WarmEjectSupported:1;
  ULONG Reserved:15;
  ULONG Address;
  ULONG UINumber;
  DEVICE_POWER_STATE DeviceState[PowerSystemMaximum];
  SYSTEM_POWER_STATE SystemWake;
  DEVICE_POWER_STATE DeviceWake;
  ULONG D1Latency;
  ULONG D2Latency;
  ULONG D3Latency;
} DEVICE_CAPABILITIES, *PDEVICE_CAPABILITIES;
```

A DEVICE_CAPABILITIES structure describes PnP and power capabilities of a device. This structure is returned in response to an IRP_MN_QUERY_CAPABILITIES IRP.

Members

Size

Specifies the size of the structure, in bytes. This field is set by the component that sends the IRP_MN_QUERY_CAPABILITIES request.

Version

Specifies the version of the structure, currently version 1. This field is set by the component that sends the IRP_MN_QUERY_CAPABILITIES request.

DeviceD1

Specifies whether the device hardware supports the D1 power state. Drivers should not change this value.

DeviceD2

Specifies whether the device hardware supports the D2 power state. Drivers should not change this value.

LockSupported

Specifies whether the device supports physical-device locking that prevents device ejection. This member pertains to ejecting the device from its slot, rather than ejecting a piece of removeable media from the device.

EjectSupported

Specifies whether the device supports software-controlled device ejection while the system is in the **PowerSystemWorking** state. This member pertains to ejecting the device from its slot, rather than ejecting a piece of removable media from the device.

Removable

Specifies whether the device can be dynamically removed from the system. If TRUE, the device is displayed in the Unplug or Eject Hardware applet, unless **SurpriseRemovalOK** is also set to TRUE.

DockDevice

Specifies whether the device is a docking peripheral.

UniqueID

Specifies whether the device supports system-wide unique IDs (that is, the concatenation of its DeviceID and its InstanceID is unique system-wide). This bit is clear if the IDs that the device supports are unique only within the scope of the bus.

SilentInstall

Specifies whether the Device Manager should suppress all installation pop-ups; except required pop-ups such as "no compatible drivers found."

RawDeviceOK

Specifies whether the driver for the underlying bus can drive the device if there is no function driver (for example, SCSI devices in pass-through mode).

SurpriseRemovalOK

Specifies whether the system should display a pop-up window if a user removes the device from the machine without first going through the Unplug or Eject Hardware applet (a "surprise-style" removal).

WakeFromD0

Specifies whether the device can respond to an external wake signal while in the D0 state. Drivers should not change this value.

WakeFromD1

Specifies whether the device can respond to an external wake signal while in the D1 state. Drivers should not change this value.

WakeFromD2

Specifies whether the device can respond to an external wake signal while in the D2 state. Drivers should not change this value.

WakeFromD3

Specifies whether the device can respond to an external wake signal while in the D3 state. Drivers should not change this value.

HardwareDisabled

When set, this flag specifies that the device's hardware is disabled.

A device's parent bus driver or a bus filter driver sets this flag when such a driver determines that the device hardware is disabled.

The PnP Manager sends one IRP_MN_QUERY_CAPABILITIES IRP right after a device is enumerated and sends another after the device has been started. The PnP Manager only checks this bit right after the device is enumerated. Once the device is started, this bit is ignored.

NonDynamic

Reserved for future use.

WarmEjectSupported

Reserved for future use.

Reserved

Reserved for system use.

Address

Specifies an address indicating where the device is located on its underlying bus.

The interpretation of this number is bus-specific. If the address is unknown or the bus driver does not support an address, the bus driver leaves this member at its default value of 0xFFFFFFFF.

The following list describes the information certain bus drivers store in the **Address** field for their child devices:

1394

Does not supply an address because the addresses are volatile. Defaults to 0xFFFFFFFF.

EISA

Slot Number (0-F).

IDE

For an IDE device, the address contains the target ID and LUN. For an IDE channel, the address is zero or one (0 = primary channel and 1 = secondary channel).

ISApnp

Does not supply an address. Defaults to 0xFFFFFFFF.

PC Card (PCMCIA)

The socket number (typically 0x00 or 0x40).

PCI

The device number in the high word and the function number in the low word.

SCSI

The target ID.

USB

The port number.

UINumber

Specifies a number associated with the device that can be displayed in the user interface.

This number is typically a user-perceived slot number, such as a number printed next to the slot on the board, or some other number that makes locating the physical device easier for the user. For buses with no such convention, or when the **UINumber** is unknown, the bus driver leaves this member at its default value of 0xFFFFFFFF.

DeviceState

An array of values indicating the most-powered device power state that the device can maintain for each system power state. The **DeviceState[PowerSystemWorking]** element of the array corresponds to the S0 system state. The entry for **PowerSystemUnspecified** is reserved for system use.

The entries in this array are based on the capabilities of the parent devnode. As a general rule, a driver should not change these values. However, if necessary, a driver can lower the value, for example, from **PowerDeviceD1** to **PowerDeviceD2**.

If the bus driver is unable to determine the appropriate device power state for a root-enumerated device, it sets **DeviceState[PowerSystemWorking]** to **PowerDeviceD0** and all other entries to **PowerDeviceD3**.

SystemWake

Specifies the least-powered system power state from which the device can signal a wake event. A value of **PowerSystemUndefined** indicates that the device cannot wake the system.

A bus driver can get this information from its parent devnode.

In general, a driver should not change this value. If necessary, however, a driver can raise the power state, for example, from **PowerSystemHibernate** to **PowerSystemS1**, to indicate that its device cannot wake the system from a hibernation state but can from a higher-powered sleep state.

DeviceWake

Specifies the least-powered device power state from which the device can signal a wake event. A value of **PowerDeviceUndefined** indicates that the device cannot signal a wake event.

D1Latency

Specifies the device's approximate worst-case latency, in 100-microsecond units, for returning the device to the **PowerDeviceD0** state from the **PowerDeviceD1** state. Set to zero if the device does not support the D1 state.

D2Latency

Specifies the device's approximate worst-case latency, in 100-microsecond units, for returning the device to the **PowerDeviceD0** state from the **PowerDeviceD2** state. Set to zero if the device does not support the D2 state.

D3Latency

Specifies the device's approximate worst-case latency, in 100-microsecond units, for returning the device to the **PowerDeviceD0** state from the **PowerDeviceD3** state. Set to zero if the device does not support the D3 state.

Include

wdm.h or *ntddk.h*

Comments

Bus drivers set the appropriate values in this structure in response to an IRP_MN_QUERY_CAPABILITIES IRP. Bus filter drivers, function drivers, and filter drivers might alter the capabilities set by the bus driver.

Drivers that send an IRP_MN_QUERY_CAPABILITIES request must initialize the **Size**, **Version**, **Address**, and **UINumber** members of this structure before sending the IRP.

See Also

IRP_MN_QUERY_CAPABILITIES

DEVICE_INTERFACE_CHANGE_NOTIFICATION

```
typedef struct _DEVICE_INTERFACE_CHANGE_NOTIFICATION {
  USHORT Version;
  USHORT Size;
  GUID Event;
  //
  // Event-specific data
  //
  GUID InterfaceClassGuid;
  PUNICODE_STRING SymbolicLinkName;
} DEVICE_INTERFACE_CHANGE_NOTIFICATION, *PDEVICE_INTERFACE_CHANGE_NOTIFICATION;
```

A device-interface-change notification structure describes a device interface that has been enabled (arrived) or disabled (removed). The PnP Manager sends this structure to a driver that registered a callback routine for notification of **EventCategoryDeviceInterfaceChange** events.

Members

Version

Specifies the version of the data structure, currently 1.

Size

Specifies the size of the structure, in bytes, including the size of the standard first three members plus the event-specific data.

Event

Specifies a GUID identifying the event: GUID_DEVICE_INTERFACE_ARRIVAL or GUID_DEVICE_INTERFACE_REMOVAL. The GUIDs are defined in *wdmguid.h.*

InterfaceClassGuid

Specifies the class of the device interface that has just been enabled or disabled.

SymbolicLinkName

Points to a Unicode string that contains the name of the symbolic link for the device interface.

Include

wdm.h or *ntddk.h*

Comments

This structure is allocated from paged memory.

See Also

HWPROFILE_CHANGE_NOTIFICATION, **IoRegisterPlugPlayNotification**, PLUGPLAY_NOTIFICATION_HEADER, TARGET_DEVICE_REMOVAL_NOTIFICATION

HWPROFILE_CHANGE_NOTIFICATION

```
typedef struct _HWPROFILE_CHANGE_NOTIFICATION {
  USHORT Version;
  USHORT Size;
  GUID Event;
  //
  // (No event-specific data)
  //
} HWPROFILE_CHANGE_NOTIFICATION, *PHWPROFILE_CHANGE_NOTIFICATION;
```

A hardware-profile-change notification structure describes an event related to a hardware profile configuration change. The PnP Manager sends this structure to a driver that registered a callback routine for notification of **EventCategoryHardwareProfileChange** events.

Members

Version

Specifies the version of the data structure, currently 1.

Size

Specifies the size of the structure, in bytes including the size of the standard first three members plus the event-specific data.

Event

Specifies a GUID identifying the event: GUID_HWPROFILE_QUERY_CHANGE, GUID_HWPROFILE_CHANGE_COMPLETE, or GUID_HWPROFILE_CHANGE_CANCELLED. The GUIDs are defined in *wdmguid.h.*

Include

wdm.h or *ntddk.h*

Comments

There is no event-specific data for a hardware-profile-change event.

See Also

DEVICE_INTERFACE_CHANGE_NOTIFICATION, **IoRegisterPlugPlayNotification**, PLUGPLAY_NOTIFICATION_HEADER, TARGET_DEVICE_REMOVAL_NOTIFICATION

LPGUID

```
typedef struct _GUID {
  ULONG Data1;
  USHORT Data2;
  USHORT Data3;
  UCHAR Data4[8];
} GUID

typedef GUID *LPGUID;
```

An LPGUID is a long pointer to a GUID.

Include

wdm.h or *ntddk.h*

Comments

A GUID is a 128-bit unique identifier.

PLUGPLAY_NOTIFICATION_HEADER

```
typedef struct _PLUGPLAY_NOTIFICATION_HEADER {
  USHORT Version;
  USHORT Size;
  GUID Event;
} PLUGPLAY_NOTIFICATION_HEADER, *PPLUGPLAY_NOTIFICATION_HEADER;
```

A PnP notification header is included at the beginning of each PnP notification structure, such as a DEVICE_INTERFACE_CHANGE_NOTIFICATION structure.

Members

Version

Specifies the version of the data structure, currently set to 1.

Size

Specifies the size of the structure, in bytes.

Event

Specifies a GUID identifying the event.

Include

wdm.h or *ntddk.h*

Comments

Drivers can cast a PnP notification structure to this type to access the **Event** field and identify the exact type of the structure.

See Also

DEVICE_INTERFACE_CHANGE_NOTIFICATION, HWPROFILE_CHANGE_NOTIFICATION, **IoRegisterPlugPlayNotification**, TARGET_DEVICE_CUSTOM_NOTIFICATION, TARGET_DEVICE_REMOVAL_NOTIFICATION

PNP_DEVICE_STATE

```
typedef ULONG PNP_DEVICE_STATE, *PPNP_DEVICE_STATE;
```

PNP_DEVICE_STATE is a bitmask of flags describing the PnP state of a device. Drivers return this structure in response to an IRP_MN_QUERY_PNP_STATE IRP.

Flags

PNP_DEVICE_DISABLED

The device is physically present but is disabled in hardware.

PNP_DEVICE_DONT_DISPLAY_IN_UI

Don't display the device in the user interface. Set for a device that is physically present but not usable in the current configuration, such as a game port on a laptop that is not usable when the laptop is undocked.

PNP_DEVICE_FAILED

The device is present but not functioning properly.

When both this flag and PNP_DEVICE_RESOURCE_REQUIREMENTS_CHANGED are set, the device must be stopped before the PnP Manager assigns new hardware resources (non-stop rebalance is not supported for the device).

PNP_DEVICE_NOT_DISABLEABLE

The device must not be disabled.

A driver sets this bit for a device that is required for proper system operation. For example, if a driver receives notification that a device is in the paging path (IRP_MN_DEVICE_USAGE_NOTIFICATION for **DeviceUsageTypePaging**), the driver calls **IoInvalidateDeviceState** and sets this flag in the resulting IRP_MN_QUERY_PNP_DEVICE_STATE IRP.

If this bit is set for a device, the PnP Manager propagates this setting to the device's parent device, its parent's parent device, and so forth.

If this bit is set for a non-enumerable device, the device cannot be disabled or uninstalled.

PNP_DEVICE_REMOVED

The device has been physically removed.

PNP_DEVICE_RESOURCE_REQUIREMENTS_CHANGED

The resource requirements for the device have changed.

Typically, a bus driver sets this flag when it has determined that it must expand its resource requirements in order to enumerate a new child device.

Include

wdm.h or *ntddk.h*

Comments

The PnP Manager queries a device's PNP_DEVICE_STATE right after starting the device by sending an IRP_MN_QUERY_PNP_DEVICE_STATE request to the device stack. In response to this IRP, the drivers for the device set the appropriate flags in PNP_DEVICE_STATE.

If any of the state characteristics change after the initial query, a driver notifies the PnP Manager by calling **IoInvalidateDeviceState**. In response to a call to **IoInvalidate-DeviceState**, the PnP Manager queries the device's PNP_DEVICE_STATE again.

If a device is marked PNP_DEVICE_NOT_DISABLEABLE, the debugger displays a DNUF_NOT_DISABLEABLE user flag for the devnode. The debugger also displays a **DisableableDepends** value that counts the number of reasons why the device cannot be disabled. This value is the sum of X+Y, where X is one if the device cannot be disabled and Y is the count of the device's child devices that cannot be disabled.

See Also

IoInvalidateDeviceState, IRP_MN_QUERY_PNP_DEVICE_STATE

TARGET_DEVICE_CUSTOM_NOTIFICATION

```
typedef struct _TARGET_DEVICE_CUSTOM_NOTIFICATION {
  USHORT Version;
  USHORT Size;
  GUID Event;
  //
  // Event-specific data
  //
  PFILE_OBJECT FileObject;
  LONG NameBufferOffset;
  UCHAR CustomDataBuffer[1];
} TARGET_DEVICE_CUSTOM_NOTIFICATION, *PTARGET_DEVICE_CUSTOM_NOTIFICATION;
```

A TARGET_DEVICE_CUSTOM_NOTIFICATION structure describes a custom device event.

Members

Version

Specifies the version of the data structure, currently 1.

Size

Specifies the size of the structure, in bytes, including the first three standard members plus the event-specific data.

Event

Specifies a GUID identifying the event. GUIDs for custom event notification are defined by the components that use this mechanism.

FileObject

Points to a file object for the device.

NameBufferOffset

Specifies the offset, in bytes, from beginning of *CustomDataBuffer* where text begins. A value of -1 indicates that there is no text.

CustomDataBuffer

A variable-length buffer, optionally containing binary data at the start of the buffer, followed by an optional text buffer (word-aligned).

Include

wdm.h or *ntddk.h*

Comments

Kernel-mode components use this structure for custom event notification: to signal a custom event (**IoReportTargetDeviceChange**[**Asynchronous**]) and when handling a custom event (in a notification callback routine).

This structure accommodates both a variable-length binary data buffer and a variable-length Unicode text buffer. The *NameBufferOffset* must indicate where the text buffer begins, so the data can be delivered in the appropriate format (ANSI or Unicode) to user-mode applications that registered for handle-based notification with **RegisterDeviceNotification**. See the *Platform SDK* documentation for information on **RegisterDeviceNotification**.

See Also

IoRegisterPlugPlayNotification, **IoReportTargetDeviceChange**, **IoReportTargetDeviceChangeAsynchronous**

TARGET_DEVICE_REMOVAL_NOTIFICATION

```
typedef struct _TARGET_DEVICE_REMOVAL_NOTIFICATION {
  USHORT Version;
  USHORT Size;
  GUID Event;
  //
  // Event-specific data
  //
  PFILE_OBJECT FileObject;
} TARGET_DEVICE_REMOVAL_NOTIFICATION, *PTARGET_DEVICE_REMOVAL_NOTIFICATION;
```

A TARGET_DEVICE_REMOVAL_NOTIFICATION structure describes a device-removal event. The PnP Manager sends this structure to a driver that registered a callback routine for notification of **EventCategoryTargetDeviceChange** events.

Members

Version

Specifies the version of the data structure, currently set to 1.

Size

Specifies the size of the structure, in bytes, including the size of the standard first three members plus the event-specific data.

Event

Specifies a GUID identifying the event: GUID_TARGET_DEVICE_QUERY_REMOVE, GUID_TARGET_DEVICE_REMOVE_COMPLETE, or GUID_TARGET_DEVICE_REMOVE_CANCELLED. These GUIDs are defined in *wdmguid.h.*

FileObject

Points to a file object for the device.

Include

wdm.h or *ntddk.h*

See Also

DEVICE_INTERFACE_CHANGE_NOTIFICATION, HWPROFILE_CHANGE_NOTIFICATION, **IoRegisterPlugPlayNotification**, TARGET_DEVICE_CUSTOM_NOTIFICATION

P A R T 2

Power Management

C H A P T E R 1

Power Management Support Routines

All drivers that support power management call **Po*Xxx*** routines. These routines are declared in *ntddk.h* and *wdm.h.*

This chapter describes the following power management routines in alphabetical order:

- **PoCallDriver**
- **PoRegisterDeviceForIdleDetection**
- **PoRegisterSystemState**
- **PoRequestPowerIrp**
- **PoSetDeviceBusy**
- **PoSetPowerState**
- **PoSetSystemState**
- **PoStartNextPowerIrp**
- **PoUnregisterSystemState**

PoCallDriver

```
NTSTATUS
  PoCallDriver (
    IN PDEVICE_OBJECT DeviceObject,
    IN OUT PIRP Irp
    );
```

PoCallDriver passes a power IRP to the next lower driver in the device stack.

Parameters

DeviceObject

Points to the driver-created device object to which the IRP is to be routed.

Irp

Points to an IRP.

Include

ntddk.h or *wdm.h*

Return Value

PoCallDriver returns STATUS_SUCCESS to indicate success. It returns STATUS_PENDING if it has queued the IRP.

Comments

Drivers call **PoCallDriver**—not **IoCallDriver**—to pass a power IRP to the next lower driver. Drivers must call **PoStartNextPowerIrp** before calling **PoCallDriver**.

A driver that requires a new IRP should call **PoRequestPowerIrp**. A driver must not allocate its own power IRP.

When passing a power IRP down to the next lower driver, the caller should use **IoSkipCurrentIrpStackLocation** or **IoCopyCurrentIrpStackLocationToNext** to set the IRP stack location, then call **PoCallDriver**. Use **IoCopyCurrentIrpStackLocationToNext** if processing the IRP requires setting a completion routine, or **IoSkipCurrentStackLocation** if no completion routine is needed.

When a device is powering up, its drivers must set completion routines to perform start-up tasks (initializing the device, restoring context, etc.) after the bus driver has set the device in the working state. Set completion routines before calling **PoCallDriver**.

When a device is powering down, its drivers rarely set completion routines; they must perform necessary power-down tasks before passing the IRP to the next lower driver. After the IRP has reached the bus driver, the device will be powered off and its drivers no longer have access to it.

Only one inrush IRP can be active in the system at a time. When passing a power-up IRP for a device that requires inrush current (i.e. the DO_POWER_INRUSH flag is set in the device object), **PoCallDriver** checks whether another inrush IRP is already active. If so, **PoCallDriver** queues the current IRP for handling after the previous IRP completes and returns STATUS_PENDING. See *Setting Device Object Flags for Power Management* in Part 3, "Power Management," in the *Plug and Play, Power Management, and Setup Design Guide* for more information on inrush IRPs.

If an IRP_MN_SET_POWER or IRP_MN_QUERY_POWER request is already active for *DeviceObject*, **PoCallDriver** queues this IRP and returns STATUS_PENDING.

On Windows® 2000, drivers that are not in the paging path (that is, the DO_POWER_PAGABLE flag is set in the device object) should call **PoCallDriver** at IRQL PASSIVE_LEVEL. Drivers that are in the paging path (DO_POWER_PAGABLE is not set in the device object) or require inrush current (DO_POWER_INRUSH is set in the device object) can call **PoCallDriver** at IRQL PASSIVE_LEVEL or DISPATCH_LEVEL.

On Windows 98, all drivers call **PoCallDriver** at IRQL PASSIVE_LEVEL.

See Also

PoRequestPowerIrp

PoRegisterDeviceForIdleDetection

```
PULONG
  PoRegisterDeviceForIdleDetection (
    IN PDEVICE_OBJECT DeviceObject,
    IN ULONG ConservationIdleTime,
    IN ULONG PerformanceIdleTime,
    IN DEVICE_POWER_STATE State
    );
```

PoRegisterDeviceForIdleDetection enables or cancels idle detection and sets idle time-out values for a device.

Parameters

DeviceObject

Points to the driver-created device object for the device. On Windows 2000, this parameter can point to a PDO or FDO. On Windows 98, this parameter must point to the PDO of the underlying device.

ConservationIdleTime

Sets the time-out value (in seconds) to apply when the system power policy optimizes for energy conservation. Specify zero to disable idle detection when conservation policy is in effect.

PerformanceIdleTime

Sets the time-out value (in seconds) to apply when the system power policy optimizes for performance. Specify zero to disable idle detection when performance policy is in effect.

State

Specifies the device power state to be requested in an IRP_MN_SET_POWER request when either *ConservationIdleTime* or *PerformanceIdleTime* has been met. Possible values are **PowerDeviceD0**, **PowerDeviceD1**, **PowerDeviceD2**, and **PowerDeviceD3**.

Include

ntddk.h or *wdm.h*

Return Value

PoRegisterDeviceForIdleDetection returns a pointer to the idle counter to indicate that idle detection has been enabled. It returns NULL to indicate that idle detection has been disabled, that an idle counter could not be allocated, or that one or both of the time-out values were invalid.

Comments

PoRegisterDeviceForIdleDetection enables drivers to use the Power Manger's idle detection mechanism. Drivers call **PoRegisterDeviceForIdleDetection** for any of the following reasons:

- To enable idle detection for the device and set initial idle time-out values
- To change the idle time-out values for a device
- To disable idle detection for a device

After enabling its device for idle detection, the driver calls **PoSetDeviceBusy** whenever its device is in use, passing the idle pointer returned by **PoRegisterDeviceForIdleDetection**.

Whenever the device satisfies the current idle time-out value, the Power Manager sends an IRP_MN_SET_POWER request to the top of the device stack, specifying device power state *State*. In response to the IRP, each driver performs any device-specific tasks required before the power state transition, then passes the IRP to the next lower driver. When the IRP reaches the bus driver, that driver puts the device in the requested lower power state and completes the IRP.

PoRegisterDeviceForIdleDetection sets time-out values for both conservation and performance. The *ConservationIdleTime* value applies when the system power policy optimizes for conservation; the *PerformanceIdleTime* value applies when the system power policy optimizes for performance. Typically, the applicable policy depends upon the power source: when running with AC power, the system optimizes for performance, and when running off a battery, the system optimizes for conservation.

Certain devices can specify time-out values of -1 to use the standard power policy time-outs for their device class. The standard time-out values provide for better system integration for supported standard device classes. At present, WDM supports this feature for devices of type FILE_DEVICE_DISK and FILE_DEVICE_MASS_STORAGE. **PoRegisterDevice-ForIdleDetection** returns NULL if -1 is specified for a device of an unsupported type.

Only one idle detection can be set per device. Subsequent calls to **PoRegisterDeviceFor-IdleDetection** change the idle detection values.

If both *ConservationIdleTime* and *PerformanceIdleTime* are zero, this routine cancels all idle detection for the device and returns NULL.

PoRegisterDeviceForIdleDetection can free a driver from the need to perform its own idle detection. However, drivers can also implement their own idle detection.

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

See Also

PoSetDeviceBusy

PoRegisterSystemState

```
PVOID
  PoRegisterSystemState (
    IN PVOID StateHandle,
    IN EXECUTION_STATE Flags
    );
```

PoRegisterSystemState registers the system as busy due to certain activity.

Parameters

StateHandle

Points to a caller-supplied memory location that can contain a registration state handle. If NULL, this is a new registration. If non-NULL, this parameter points to a handle returned by a previous call to **PoRegisterSystemState**.

Flags

Indicates the type of activity. Possible values are one or more of the following:

ES_SYSTEM_REQUIRED

The system is not idle, regardless of apparent load.

ES_DISPLAY_REQUIRED

Use of the display is required.

ES_USER_PRESENT

A user is present.

ES_CONTINUOUS

The settings are continuous and should remain in effect until explicitly changed.

Include

ntddk.h or *wdm.h*

Return Value

PoRegisterSystemState returns a handle to be used later to change or unregister the system busy state. It returns NULL if the handle could not be allocated.

Comments

PoRegisterSystemState registers the system busy state as indicated by the flags. The registration persists until the caller explicitly changes it with another call to **PoRegister-SystemState** or cancels it with a call to **PoUnregisterSystemState**.

The *Flags* parameter specifies the type of activity in progress. Drivers can specify any combination of the flags.

Setting ES_CONTINUOUS makes the busy state persist until a driver explicitly changes or cancels it by calling **PoRegisterSystemState** or **PoUnregisterSystemState**.

A driver can set the system busy state to request that the Power Manager avoid system power state transitions out of the system working state (S0) while driver activity is occurring. Note, however, that under some circumstances (such as a critically low battery) the Power Manager may override this request and put the system to sleep anyway.

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

See Also

PoSetSystemState, PoUnregisterSystemState

PoRequestPowerIrp

```
NTSTATUS
  PoRequestPowerIrp (
    IN PDEVICE_OBJECT DeviceObject,
    IN UCHAR MinorFunction,
    IN POWER_STATE PowerState,
    IN PREQUEST_POWER_COMPLETE CompletionFunction,
    IN PVOID Context,
    OUT PIRP *Irp OPTIONAL
    );
```

PoRequestPowerIrp allocates a power IRP and sends it to the top driver in the device stack for the specified device.

Parameters

DeviceObject

Points to the target device object for the IRP. On Windows 2000, this parameter can point to a PDO or FDO. On Windows 98, this parameter must point to the PDO of the underlying device.

MinorFunction

Specifies one of the following minor power IRP codes: IRP_MN_QUERY_POWER, IRP_MN_SET_POWER, or IRP_MN_WAIT_WAKE.

PowerState

Specifies a power state to pass in the IRP. For IRP_MN_SET_POWER and IRP_MN_QUERY_POWER, specify the requested new device power state. Possible values are enumerators of the DEVICE_POWER_STATE type.

For IRP_MN_WAIT_WAKE, specify the lowest (least-powered) system power state from which the device should be allowed to wake the system Possible values are enumerators of the SYSTEM_POWER_STATE type.

CompletionFunction

Points to the caller's PowerCompletion callback to be called when the IRP has completed. The callback is declared as follows:

```
VOID
(*PREQUEST_POWER_COMPLETE) (
    IN PDEVICE_OBJECT DeviceObject,
    IN UCHAR MinorFunction,
    IN POWER_STATE PowerState,
    IN PVOID Context,
    IN PIO_STATUS_BLOCK IoStatus
    );
```

The callback parameters are as follows:

DeviceObject

Points to the target device object for the completed power IRP.

MinorFunction

Specifies the minor function code in the power IRP.

PowerState

Specifies the device power state passed to **PoRequestPowerIrp**.

Context

Points to the context passed to **PoRequestPowerIrp**.

IoStatus

Points to the **IoStatus** block in the completed IRP.

Context

Points to a caller-supplied context to be passed through to the PowerCompletion callback. When the caller requests a device set-power IRP in response to a system set-power IRP, the *Context* should contain the system set-power IRP that triggered the request.

Irp

Points to a caller-supplied variable in which this routine returns a pointer to the IRP it allocates. This parameter can be NULL.

Include

ntddk.h or *wdm.h*

Return Value

PoRequestPowerIrp returns one of the following:

STATUS_PENDING

The IRP has been sent.

STATUS_INSUFFICIENT_RESOURCES

The routine could not allocate the IRP.

STATUS_INVALID_PARAMETER_2

MinorFunction does not signify a valid minor power IRP code.

Comments

A driver calls **PoRequestPowerIrp**—not **IoAllocateIrp**—to allocate and send a power IRP that has minor IRP code IRP_MN_SET_POWER, IRP_MN_QUERY_POWER, or IRP_MN_WAIT_WAKE. (A driver must call **IoAllocateIrp** to send a power IRP with minor IRP code IRP_MN_POWER_SEQUENCE.)

A device power policy owner calls this routine to send a wait/wake, query, or set-power IRP.

PoRequestPowerIrp allocates a device power IRP and sends it to the top of the device stack for the device. After the bus driver and all other drivers have completed the IRP, and the I/O Manager has called all IoCompletion routines set by drivers as they passed the IRP down the device stack, the *CompletionFunction* is called with the given *Context*.

The *CompletionFunction* performs any additional tasks the sender of the IRP requires after all other drivers have completed the IRP. It need not free the IRP; the Power Manager does that. On Windows 98, the *CompletionFunction* is always called at IRQL PASSIVE_LEVEL, and drivers must complete IRPs at IRQL PASSIVE_LEVEL. On Windows 2000, the *CompletionFunction* can be called at IRQL PASSIVE_LEVEL or DISPATCH_LEVEL.

A device power policy owner calls **PoRequestPowerIrp** to send a device query- or set-power IRP when it receives a system query- or set-power IRP. The driver should set an IoCompletion routine in the system IRP and pass the system IRP to the next lower driver. The IoCompletion routine calls **PoRequestPowerIrp** to send the device IRP, passing the system IRP in the *Context* parameter. The *Context* parameter is subsequently passed to the *CompletionFunction* for the device IRP. In the *CompletionFunction*, the driver can complete the system IRP. See *Sending IRP_MN_QUERY_POWER or IRP_MN_SET_POWER for Device Power States* and *Wait/Wake Callback Routines* for details.

If the driver supports the GUID_POWER_DEVICE_ENABLE control, the driver should use the Boolean value of the control to determine whether to dynamically power itself on and off while the system is in the **PowerSystemWorking** (S0) state. See the Volume 2 of the *Windows 2000 Driver Development Reference* for more information on IRP_MJ_SYSTEM_CONTROL IRPs, which send requests with minor code IRP_MN_WMI to inform drivers of the value of the GUIDs.

Drivers can use the returned *Irp* to cancel an IRP_MN_WAIT_WAKE IRP. Drivers requesting other power IRPs can pass NULL for this parameter.

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

See Also

PoStartNextPowerIrp, IRP_MN_SET_POWER, IRP_MN_QUERY_POWER, IRP_MN_WAIT_WAKE

PoSetDeviceBusy

```
VOID
  PoSetDeviceBusy(
    PULONG IdlePointer
    );
```

PoSetDeviceBusy notifies the Power Manager that the device associated with *IdlePointer* is busy.

Parameters

IdlePointer

Specifies an idle pointer previously returned by **PoRegisterDeviceForIdleDetection**.

Include

ntddk.h or *wdm.h*

Comments

A driver uses **PoSetDeviceBusy** along with **PoRegisterDeviceForIdleDetection** to enable system idle detection for its device. If a device that is registered for idle detection becomes idle, the Power Manager sends an IRP_MN_SET_POWER IRP to put the device in a requested sleep state.

PoSetDeviceBusy reports that the device is busy, so that the Power Manager can restart its idle countdown. If the device is not powered up, **PoSetDeviceBusy** does not change its state. That is, it does not cause the system to send a power-on request.

A driver should call **PoSetDeviceBusy** on every I/O request.

PoSetDeviceBusy can be called from any IRQL.

See Also

PoRegisterDeviceForIdleDetection

PoSetPowerState

```
POWER_STATE
  PoSetPowerState (
    IN PDEVICE_OBJECT DeviceObject,
    IN POWER_STATE_TYPE Type,
    IN POWER_STATE State
    );
```

PoSetPowerState notifies the system of a device's power state.

Parameters

DeviceObject

Points to the target device object.

Type

Indicates whether to set a system or a device power state. Drivers must specify **DevicePowerState**.

State

Specifies the power state to be set. Drivers must specify an enumerator of DEVICE_POWER_STATE: **PowerDeviceD0**, **PowerDeviceD1**, **PowerDeviceD2**, or **PowerDeviceD3**.

Include

ntddk.h or *wdm.h*

Return Value

On Windows 2000, **PoSetPowerState** returns the previous power state. On Windows 98, **PoSetPowerState** returns the state passed in *State*.

Comments

PoSetPowerState notifies the Power Manager of the new power state for a device. A driver should call **PoSetPowerState** every time its device changes power state.

A driver calls this routine after receiving a device set-power IRP and before calling **PoStartNextPowerIrp**. When starting a device (that is, when handling a PnP IRP_MN_START_DEVICE request), the driver should call **PoSetPowerState** to notify the Power Manager that the device is in the D0 state.

If the device is powering down, the driver must call **PoSetPowerState** before leaving the D0 state. In addition, the driver must be able to process client requests before **PoSetPowerState** returns.

If the device is powering up, the driver must call **PoSetPowerState** after the device is successfully put into the D0 state.

Callers of **PoSetPowerState** must be running at IRQL<DISPATCH_LEVEL except when setting state to D0. When setting state to D0, callers can be running at IRQL<= DISPATCH_LEVEL.

See Also

PoStartNextPowerIrp

PoSetSystemState

```
VOID
  PoSetSystemState (
    IN EXECUTION_STATE Flags
    );
```

Drivers call **PoSetSystemState** to indicate that the system is active.

Parameters

Flags

Specifies the system activity. Possible values are one or more of the following:

ES_SYSTEM_REQUIRED

The system is not idle, regardless of apparent load.

ES_DISPLAY_REQUIRED

Use of the display is required.

ES_USER_PRESENT

A user is present.

Include

ntddk.h or *wdm.h*

Comments

A driver calls **PoSetSystemState** to set flags indicating that system activity is occurring. Unlike **PoRegisterSystemState**, this routine does not allow the driver to set a persistent busy state.

The *Flags* parameter specifies the type of activity occurring. Drivers can specify any combination of the flags.

Drivers can set the system busy state to request that the system avoid leaving of the working state while driver activity is occurring. Note, however, that under some circumstances (such as a critically low battery) the Power Manager may override this request and put the system to sleep anyway.

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

See Also

PoRegisterSystemState, **PoUnregisterSystemState**

PoStartNextPowerIrp

```
VOID
  PoStartNextPowerIrp(
    IN PIRP Irp
    );
```

PoStartNextPowerIrp informs the Power Manager that the driver is ready to handle the next power IRP.

Parameters

Irp

Points to an IRP in which the major function code is IRP_MJ_POWER.

Include

ntddk.h or *wdm.h*

Comments

PoStartNextPowerIrp must be called by every driver in the device stack.

Calling this routine indicates that the driver is finished with the previous power IRP, if any, and is ready to handle the next power IRP. It must be called for every power IRP.

Although power IRPs are completed only once, typically by the bus driver for a device, each driver in the device stack must call **PoStartNextPowerIrp** as the IRP travels down or back up the stack. Even if a driver fails the IRP, it must nevertheless call **PoStartNextPowerIrp** to inform the Power Manager that it is ready to handle another power IRP.

The driver must call **PoStartNextPowerIrp** while the current IRP stack location points to the current driver. Therefore, this routine must be called before **IoCompleteRequest**, **IoSkipCurrentIrpStackLocation**, and **PoCallDriver**. Note, however, that a driver can call **PoStartNextPowerIrp** from its IoCompletion routine associated with the IRP or from the callback routine it passed to **PoRequestPowerIrp**.

Bus drivers must call **PoStartNextPowerIrp** before completing each IRP.

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

See Also

PoCallDriver, **PoStartNextPowerIRP**

PoUnregisterSystemState

```
VOID
  PoUnregisterSystemState (
    IN PVOID StateHandle
    );
```

PoUnregisterSystemState cancels a system state registration created by **PoRegister-SystemState**.

Parameters

StateHandle

Specifies a handle previously returned by **PoRegisterSystemState**.

Include

ntddk.h or *wdm.h*

Comments

This routine cancels a system busy state registration established by **PoRegisterSystemState** and frees the associated *StateHandle*.

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

See Also

PoRegisterSystemState

CHAPTER 2

I/O Request for Power Management

All power IRPs have the major code IRP_MJ_POWER and one of the following minor codes, indicating a specific power management request:

- IRP_MN_SET_POWER
- IRP_MN_QUERY_POWER
- IRP_MN_WAIT_WAKE
- IRP_MN_POWER_SEQUENCE

For each power IRP a driver receives, it must call **PoStartNextPowerIrp** to indicate to the Power Manager that it is ready to handle the next power IRP.

After calling **PoStartNextPowerIrp**, the driver must use **PoCallDriver** to pass the power IRP down the device stack. Power IRPs are typically completed by the bus driver for the device, and therefore must be passed all the way down the device stack.

This chapter provides reference information for the individual IRPs in alphabetical order. See the *Plug and Play, Power Management, and Setup Design Guide* for more information on when the IRPs are sent and how drivers should handle them, along with a general introduction to power management and terminology.

IRP_MN_POWER_SEQUENCE

When Sent

The Power Manager cannot send this IRP. A driver sends this IRP as an optimization to determine whether its device actually entered a specific power state. The IRP is optional.

To send this IRP, a driver must call **IoAllocateIrp** to allocate the IRP, specifying the major IRP code IRP_MJ_POWER and minor IRP code IRP_MN_POWER_SEQUENCE, and then call **PoCallDriver** to pass the IRP to the next lower driver. Senders of this IRP must be running at IRQL <= DISPATCH_LEVEL.

Input

None.

Output

Parameters.PowerSequence points to a POWER_SEQUENCE structure with the following members:

SequenceD1

Number of times the device has been in power state D1 or lower.

SequenceD2

Number of times the device has been in power state D2 or lower.

SequenceD3

Number of times the device has been in power state D3.

The sequence values track the minimum number of times a device has been in the corresponding power state or a lower power state.

The bus driver increments the values in **SequenceD1**, **SequenceD2**, and **SequenceD3** at least each time the device enters in the corresponding power state or a lower power state.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS to indicate that it has returned the requested information, or STATUS_NOT_IMPLEMENTED to indicate that it does not support this IRP.

Operation

This IRP returns the power sequence values for a device. Bus drivers can optionally handle it; function and filter drivers can optionally send it.

For a device that takes a long time to change state, this IRP provides a useful optimization. Every time the device changes its power state, its bus driver increments the sequence value corresponding to that power state. The bus driver initializes the sequence values at boot time and continually increments them thereafter; they need not be reset to zero.

A device policy owner can send this IRP once to get the sequence values before shutting off the device and once again to get new values when restoring power to the device. By comparing the two sets of values, the driver can determine whether the device actually entered the lower-powered state. If the device did not lose power, the driver can avoid a time-consuming reinitialization when the device returns to the D0 state.

For example, if the device takes a long time to restore power upon reaching the D2 state, the driver can store the **SequenceD2** value before it sets the device state to D2 or lower. Later, when power is being restored to the device, the driver can compare the new **SequenceD2** value with its stored value to determine whether the device state actually dropped below D2. If the values match, the device did not actually enter a D2 or lower state, and the driver can avoid reinitializing the device.

IRP_MN_QUERY_POWER

When Sent

The Power Manager or a device power policy owner sends this IRP to determine whether it can change the system or device power state, typically to go to sleep. A driver must call **PoRequestPowerIrp** to allocate and send this IRP.

The Power Manager sends this IRP at IRQL PASSIVE_LEVEL to device stacks that set the DO_POWER_PAGABLE flag in the PDO.

The Power Manager can send the IRP at IRQL DISPATCH_LEVEL if the DO_POWER_INRUSH flag is set. Such drivers cannot directly or indirectly access any paged code or data.

Input

Parameters.Power.Type specifies the type of power state being set, either **SystemPowerState** or **DevicePowerState**.

Parameters.Power.State specifies the power state itself, as follows:

- If **Parameters.Power.Type** is **SystemPowerState**, the value is an enumerator of the SYSTEM_POWER_STATE type.
- If **Parameters.Power.Type** is **DevicePowerState**, the value is an enumerator of the DEVICE_POWER_STATE type.
- **Parameters.Power.ShutdownType** specifies additional information about the requested transition. Possible values are enumerators of the POWER_ACTION type.

Output

None.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS to indicate that the device can enter the requested state. A driver sets any appropriate failure status to indicate that it cannot enter the requested state.

Operation

The parameters for IRP_MN_QUERY_POWER are identical to those for IRP_MN_SET_POWER. Rather than notifying drivers of an irrevocable change to the power state, however, IRP_MN_QUERY_POWER queries whether the system or a device can enter a particular power state.

A driver must not change the power state of its device in response to an IRP_MN_QUERY_POWER request.

IRP_MN_QUERY_POWER for a System Power State

The Power Manager sends this IRP to ensure that it can change the system power state without disrupting work, such as dropping network connections.

Whenever possible, the Power Manager queries before sending IRP_MN_SET_POWER to request a system sleep state. However, under some conditions (such as the user pressing the Power Off button or a battery expiring), the Power Manager might issue an IRP_MN_SET_POWER request without first querying. The Power Manager queries only for sleep states; it never queries before returning to the working state.

When a driver receives a system power query IRP, it should fail the IRP if it cannot support any of the device states that are valid for the queried system state. See **DeviceState** for details. Otherwise, the driver should pass the IRP to the next lower driver. The bus driver completes the IRP.

When a device power policy owner receives a system power query IRP, it should set an IoCompletion routine in the IRP before passing it down. In the IoCompletion routine, it should send an IRP_MN_QUERY_POWER for a device state that is valid for the queried system state. See *Handling a System Query Power IRP in a Device Power Policy Owner* for details.

When the IRP specifies **PowerSystemShutdown** (S5), the value at **Parameters.Power.ShutdownType** provides a reason for the shutdown. The **ShutdownType** tells the driver whether the system is resetting (**PowerActionShutdownReset**) or powering off indefinitely to reboot later (**PowerActionShutdownOff**). For drivers of most devices, the difference is inconsequential. However, for certain devices, such as a video streaming device that performs DMA, a driver might choose to power down its device when the system is resetting, thus stopping any ongoing I/O.

On Microsoft® Windows® 2000 systems, the value at **ShutdownType** can also be **PowerActionShutdown**. In this case, the driver cannot tell what type of shutdown is requested and should therefore proceed as for a reset.

If a driver fails a IRP_MN_QUERY_POWER request for a system power state, the Power Manager typically responds by issuing an IRP_MN_SET_POWER IRP. Usually, this IRP will reaffirm the current system state. However, it is possible that drivers might receive an

IRP_MN_SET_POWER to the queried state or to some other intermediate state. Drivers should be prepared to handle these situations.

IRP_MN_QUERY_POWER for a Device Power State

A device power policy owner sends this IRP down its stack in response to a system IRP_MN_QUERY_POWER request.

If a driver can put its device in the requested device state, it sets **IoStatus.Status** to STATUS_SUCCESS and passes the IRP down to the next lower driver, and so forth until the IRP reaches the bus driver. If any driver in the stack must fail the IRP, that driver should complete the IRP immediately by calling **IoCompleteRequest** and returning a failure status. Drivers that fail the IRP do not pass it further down the stack.

By returning STATUS_SUCCESS, the driver guarantees that it will not start any operation that would change its ability to set the requested power state. The driver should queue any IRPs that require such operations until it completes a set-power IRP that returns the device to an acceptable power state.

On Windows 2000 systems, when the IRP specifies **PowerDeviceD1**, **PowerDeviceD2**, or **PowerDeviceD3**, the value at **Parameters.Power.ShutdownType** provides information about the current system power IRP, if a system power IRP is active. In this case, the value at **ShutdownType** indicates the currently requested system power state, or **PowerActionNone** if a system request is not outstanding. On Windows 98, this field always contains **PowerActionNone** when the IRP requests a device power state.

See Also

IRP_MN_SET_POWER, **PoRequestPowerIrp**

IRP_MN_SET_POWER

When Sent

Either the system Power Manager or a device power policy owner can send this IRP.

The Power Manager sends this IRP to notify drivers of a change to the system power state. If a driver has registered its device for idle detection, the Power Manager sends this IRP to change the power state of an idle device.

A driver that owns power policy sends this IRP to set the device power state for its device. A driver must call **PoRequestPowerIrp** to send this IRP.

The Power Manager sends this IRP at IRQL PASSIVE_LEVEL to device stacks that set the DO_POWER_PAGABLE flag in the PDO. Drivers in such stacks can touch paged code or data to complete the request.

The Power Manager can send the IRP at IRQL DISPATCH_LEVEL if the DO_POWER_INRUSH flag is set. Such drivers cannot directly or indirectly access any paged code or data.

Input

Parameters.Power.Type specifies the type of power state being set, either **SystemPowerState** or **DevicePowerState**.

Parameters.Power.State specifies the power state itself, as follows:

- If **Parameters.Power.Type** is **SystemPowerState**, the value is an enumerator of the SYSTEM_POWER_STATE type.
- If **Parameters.Power.Type** is **DevicePowerState**, the value is an enumerator of the DEVICE_POWER_STATE type.
- **Parameters.Power.ShutdownType** specifies additional information about the requested transition. Possible values are enumerators of the POWER_ACTION type.

Output

Parameters.Power.SystemContext is reserved for system use.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to STATUS_SUCCESS to indicate that the device has entered the requested state.

A driver must not fail this IRP.

Operation

The Power Manager or a driver can request an IRP_MN_SET_POWER IRP. The Power Manager sends this IRP for one of the following reasons:

- To notify drivers of a change to the system power state
- To change the power state of a device for which the Power Manager is performing idle detection

A driver that owns device power policy sends IRP_MN_SET_POWER to change the power state of its device.

At any given time, the system allows only one such IRP to be active for each device object.

Each driver must pass each power IRP down to the next-lower driver using the **PoCallDriver** routine. The **PoCallDriver** interface is similar to that of **IoCallDriver**, except that the power management subsystem might delay the IRP before passing it on to the next driver. For example, delays can occur on a **PowerDeviceD0** request if the device requires inrush current and therefore must be powered up serially with another such device.

IRP_MN_SET_POWER for System Power States

Only the Power Manager can send a system set-power IRP.

Whenever possible, the Power Manager sends IRP_MN_QUERY_POWER before sending IRP_MN_SET_POWER to request a system sleep state. However, under some conditions (such as the user pressing the Power Off button or a battery expiring), the Power Manager might issue IRP_MN_SET_POWER without first querying. The Power Manager queries only for sleep states; it never queries before powering up.

The IRP_MN_SET_POWER request is sent to the top driver in the device stack for a device. The top driver passes the IRP down to the next lower driver and so forth until the IRP reaches the bus driver, which must complete the IRP. A filter driver typically does not need to act on a system set-power IRP, other than to pass it on. The device power policy owner, however, sets an IoCompletion routine before passing down the IRP; in the IoCompletion routine, it sends an IRP_MN_SET_POWER request for a device power IRP. See *Handling a System Set-Power IRP in a Device Power Policy Owner* for details.

A system set-power IRP informs drivers that a change to the system power state is imminent and the drivers must prepare for it. However, a driver should not change the power state of its device until it receives an IRP_MN_SET_POWER for a *device* power state. For example, a transition to a sleeping state might require that a driver power off its device. The driver must complete this IRP in a timely manner, but it can wait until the device is ready to enter the new state. In general, drivers should avoid any delay that a typical user would find noticeably slow. For example, a driver could delay a system state change to flush cached disk or network data, but should not keep a network connection alive or format a tape.

The value at **Parameters.Power.ShutdownType** provides additional information about the pending actions. When the IRP specifies **PowerSystemShutdown** (S5), a driver can determine whether the system is resetting (**PowerActionShutdownReset**) or powering off indefinitely to reboot later (**PowerActionShutdownOff**). For drivers of most devices, the difference is inconsequential. However, for certain devices, such as video streaming devices, a driver might power off the device in order to stop I/O when the system is resetting.

On Windows 2000 systems, the value at **ShutdownType** can also be **PowerAction-Shutdown**. In this case, the driver cannot tell what type of shutdown is requested and should therefore proceed as for a reset.

Device Power States

A driver must set the device into the requested state before completing the IRP.

When the IRP requests a transition to a lower power state, drivers must handle the IRP as it travels down the device stack, saving any context the driver will need to restore the device to the working state. When the bus driver receives the IRP, it does the same, then sets the device into the requested power state, calls **PoSetPowerState** to notify the Power Manager, starts the next power IRP (**PoStartNextPowerIrp**), and completes the device power IRP.

The driver must complete this IRP in a timely manner. In general, drivers should avoid any delay that a typical user would find noticeably slow. For example, a driver could delay a system state change to flush cached disk or network data, but should not keep a network connection alive or format a tape. See *Passing Power IRPs* for more information.

On Windows 2000 systems, if the IRP specifies **PowerDeviceD1**, **PowerDeviceD2**, or **PowerDeviceD3**, and a system set-power IRP is active, the value at **Parameters.Power. ShutdownType** provides information about the system IRP.

Drivers of devices on the hibernate path should inspect this value. If the IRP requests **PowerDeviceD3** and **ShutdownType** is **PowerActionHibernate**, such a driver should save any context required to restore the device, but should not power down the device; the device will enter the D3 state when the machine loses power.

On Windows 2000 systems, drivers should not rely on the value at **ShutdownType** if the requested power state is **PowerDeviceD0**.

On Windows 98, if the IRP requests a device power state, the **ShutdownType** is always **PowerActionNone**.

The driver that determines when to power down a device varies depending on the device class.

The driver that determines when to power up a device is almost always a driver that accesses the device registers. The driver must verify that the device is in the D0 state before accessing the device's hardware registers. If the device is not in the D0 state, the driver must call **PoRequestPowerIrp** to send an IRP to power up the device. A driver cannot access its device unless the device is in the D0 state.

When a driver receives a set-power IRP for device state D0, it sets an IoCompletion routine and passes the IRP to the next lower driver.

When the IRP reaches the bus driver, that driver applies (or resets) power to the device, completes the IRP, and calls **PoSetPowerState** to inform the Power Manager of the new power state for the device.

After the bus driver completes the power-up IRP, function and filter drivers handle the IRP in their IoCompletion routines as it travels back up the device stack. In the IoCompletion routine, each driver restores or reinitializes its device context and performs any other required start-up tasks.

See *Handling IRP_MN_SET_POWER for Device Power States* for details.

See Also

PoCallDriver, **PoStartNextPowerIrp**, **PoSetPowerState**, **PoRequestPowerIrp**, **PoRegisterDeviceForIdleDetection**

IRP_MN_WAIT_WAKE

When Sent

A driver that owns power policy targets this IRP to its PDO to enable its device to awaken in response to an external event, such as an incoming phone call. A driver must call **PoRequestPowerIrp** to send this IRP.

As a general rule, a driver should send this IRP as soon as it determines that its device should be enabled for wake-up. Consequently, drivers for most such devices send this IRP after powering on their devices and before completing the IRP_MN_START_DEVICE request.

However, a driver can send the IRP any time the device is in the working state (**PowerDeviceD0**). The device stack must not be in transition; that is, a driver should not send an IRP_MN_WAIT_WAKE while any other power IRP is active in its device stack.

A wait/wake IRP does not change the power state of the system or of a device. It simply enables a wake-up signal from the device. When the wake-up signal arrives, the policy owner must call **PoRequestPowerIrp** to send a set-power IRP to explicitly return its device to D0.

The driver must be running at IRQL PASSIVE_LEVEL to send this IRP. However, the IRP can be completed at IRQL DISPATCH_LEVEL.

Input

Parameters.WaitWake.SystemWake contains the lowest (least-powered) system power state from which the device should be allowed to awaken the system.

Output

None.

I/O Status Block

A driver sets **Irp->IoStatus.Status** to one of the following:

STATUS_PENDING

The driver received the IRP and is waiting for the device to signal wake-up.

STATUS_INVALID_DEVICE_STATE

The device is in a less-powered state than the **DeviceWake** state specified in the DEVICE_CAPABILITIES structure for the device, or the device cannot awaken the system from the **SystemWake** state passed in the IRP.

STATUS_NOT_SUPPORTED

The device does not support wake-up.

STATUS_DEVICE_BUSY

An IRP_MN_WAIT_WAKE request is already pending and must be completed or canceled before another IRP_MN_WAIT_WAKE request can be issued.

STATUS_SUCCESS

The device has signaled a wake event.

STATUS_CANCELED

The IRP has been canceled.

If a driver must fail this IRP, it completes the IRP immediately and does not pass the IRP to the next lower driver.

Operation

A driver sends IRP_MN_WAIT_WAKE for either of two reasons:

1. To enable its device to awaken a sleeping system in response to an external wake-up signal.
2. To enable its device to awaken from a device sleep state in response to an external wake-up signal.

The IRP must be passed down the device stack to the bus driver for the device, which calls **IoMarkIrpPending** and returns STATUS_PENDING from its DispatchPower routine. The IRP remains pending until a wake-up signal occurs or until the driver that sent the IRP cancels it.

A driver can have only one wait/wake IRP pending at a time. A driver that enumerates more than one child PDO must fail any wait/wake request that arrives while it already has such an IRP pending. However, the driver should keep an internal count of wait/wake IRPs, incrementing the count each time it receives a request and decrementing the count each time it completes a request. If the count is nonzero after it has completed a wait/wake IRP, the driver should send another wait/wake IRP to its device stack to "rearm" itself for wake-up. See *Understanding the Path of Wait/Wake IRPs Through a Device Tree* for details.

Each driver sets an IoCompletion routine as the IRP travels down the device stack. When the device signals a wake-up event, the bus driver services the wake-up signal and completes the IRP, returning STATUS_SUCCESS. The I/O Manager then calls the IoCompletion routine of the next higher driver, and so on up the device stack.

When a driver sends a wait/wake IRP, it should specify a callback routine in the **Po-RequestPowerIrp** call. In the callback routine, the driver typically services the device. The power policy owner for the device must call **PoRequestPowerIrp** to send an IRP_MN_SET_POWER for device state D0.

A driver that acts as the bus driver for one device and the policy owner for a parent device requests an IRP_MN_WAIT_WAKE IRP for the parent's PDO whenever it has an outstanding IRP_MN_WAIT_WAKE request from one or more of its child PDOs. For example, the bus driver for a USB device acts as the policy owner for the USB hub controller. In its role as policy owner, the driver sends a wait/wake IRP to the hub PDO when it receives its first wait/wake IRP from a device PDO. When the IRP completes, this same driver must determine which USB device signaled the wake-up event. If additional child device stacks have also sent wait/wake IRPs, the driver must send its own device stack a wait/wake IRP to "rearm" it for wait/wake on those children.

To cancel an IRP_MN_WAIT_WAKE, a driver calls **IoCancelIrp**. Only the driver that originated the IRP can cancel it. A driver cancels a pending IRP_MN_WAIT_WAKE when any of the following occurs:

- The driver receives a PnP IRP that stops or removes the device.
- The system is going to sleep and the device wake signal must not awaken it.

Drivers can optionally support the WMI GUID_DEVICE_WAKE_ENABLE control, which allows the user to choose whether the device can wake a sleeping system. The user interface presents this control if the driver and PDO support it, as determined by querying the device capabilities.

When the user changes the DEVICE_WAKE_ENABLE setting, the driver receives a system control IRP (IRP_MJ_SYSTEM_CONTROL) with minor IRP code IRP_MN_WMI. See Volume 2 of the *Windows 2000 Driver Development Reference* for details.

See Also

PoRequestPowerIrp

C H A P T E R 3

Battery Class Driver Routines

The battery class driver exports the following routines for use by miniclass drivers:

- **BatteryClassInitializeDevice**
- **BatteryClassIoctl**
- **BatteryClassStatusNotify**
- **BatteryClassUnload**

These routines are declared in *batclass.h*.

BatteryClassInitializeDevice

```
NTSTATUS
  BatteryClassInitializeDevice (
    IN PBATTERY_MINIPORT_INFO MiniportInfo,
    IN PVOID *ClassData
    );
```

BatteryClassInitializeDevice initializes a new battery device with the class driver.

Parameters

MiniportInfo

Points to a BATTERY_MINIPORT_INFO structure, defined as follows:

```
typedef struct {
    USHORT  MajorVersion;
    USHORT  MinorVersion;
    PVOID  Context;
    BCLASS_QUERY_TAG  QueryTag;
    BCLASS_QUERY_INFORMATION  QueryInformation;
    BCLASS_SET_INFORMATION  SetInformation;
    BCLASS_QUERY_STATUS  QueryStatus;
    BCLASS_SET_STATUS_NOTIFY  SetStatusNotify;
    BCLASS_DISABLE_STATUS_NOTIFY  DisableStatusNotify;
    PDEVICE_OBJECT  Pdo;
    PUNICODE_STRING  DeviceName;
} BATTERY_MINIPORT_INFO, *PBATTERY_MINIPORT_INFO;
```

MajorVersion
Specifies the major version number of the battery class driver. Miniclass drivers should specify BATTERY_CLASS_MAJOR_VERSION.

MinorVersion
Specifies the minor version number of the battery class driver. Miniclass drivers should specify BATTERY_CLASS_MINOR_VERSION.

Context
Points to the context area allocated by the miniclass driver.

QueryTag
Specifies the entry point of the miniclass driver's BatteryMiniQueryTag routine.

QueryInformation
Specifies the entry point of the miniclass driver's BatteryMiniQueryInformation routine.

SetInformation
Specifies the entry point of the miniclass driver's BatteryMiniSetInformation routine.

QueryStatus
Specifies the entry point of the miniclass driver's BatteryMiniQueryStatus routine.

SetStatusNotify
Specifies the entry point of the miniclass driver's BatteryMiniSetStatusNotify routine.

DisableStatusNotify
Specifies the entry point of the miniclass driver's BatteryMiniDisableStatusNotify routine.

Pdo
Points to the PDO for the battery device.

DeviceName
Points to a UNICODE string, and should be NULL.

ClassData
Points to a location at which **BatteryClassInitializeDevice** returns a handle to be used in subsequent calls to **BatteryClass*Xxx*** routines.

Return Value

BatteryClassInitializeDevice returns STATUS_SUCCESS or, possibly, STATUS_INSUFFICIENT_RESOURCES if not enough memory is available to store the battery miniclass data.

Comments

Battery miniclass drivers must call **BatteryClassInitializeDevice** to register each battery device and to pass data about the device and the miniclass driver to the battery class driver.

This routine should be called as part of the device initialization, typically from the miniclass driver's AddDevice routine.

The **Context** member of the BATTERY_MINIPORT_INFO structure points to an area where the class and miniclass drivers maintain information about the battery device and its drivers. The context area typically contains the pageable device extension from the FDO and can also include other information at the discretion of the driver writer.

The class driver passes a pointer to the context area in calls to the BatteryMiniXxx routines. In their BatteryMiniXxx routines, miniclass drivers should read and write the device extension data through the passed-in pointer.

Miniclass drivers must supply entry points for all the BatteryMini*Xxx* routines.

See Also

BatteryMiniDisableStatusNotify, **BatteryMiniQueryInformation**, **BatteryMiniQueryStatus**, **BatteryMiniQueryTag**, **BatteryMiniSetInformation**, **BatteryMiniSetStatusNotify**

BatteryClassIoctl

```
NTSTATUS
  BatteryClassIoctl (
    IN PVOID ClassData,
    IN PIRP Irp
    );
```

BatteryClassIoctl handles system-defined battery IOCTLs.

Parameters

ClassData

Points to a battery class handle previously returned by **BatteryClassInitializeDevice**.

Irp

Points to the IRP containing the IOCTL to be handled.

Return Value

BatteryClassIoctl returns STATUS_SUCCESS when it satisfies the request and completes the IRP. It returns STATUS_NOT_SUPPORTED for all IRPs other than device control IRPs that specify battery IOCTLs.

Comments

BatteryClassIoctl handles and completes device control IRPs intended for the battery. Such IRPs have one of the following I/O control codes:

- IOCTL_BATTERY_QUERY_INFORMATION
- IOCTL_BATTERY_QUERY_STATUS
- IOCTL_BATTERY_QUERY_TAG
- IOCTL_BATTERY_SET_INFORMATION

The standard battery IOCTLs correspond to the miniclass driver's BatteryMiniXxx support routines.

When the miniclass driver is called with an IRP_MJ_DEVICE_CONTROL request, it should determine whether the IRP contains any private IOCTL defined by the miniclass driver. If so, the miniclass driver should satisfy the request, complete the IRP, and return.

If the IRP contains a public IOCTL, the driver should pass the IRP to the class driver's **BatteryClassIoctl** routine. This routine examines the IRP, determines whether it applies to the caller's battery device, and if so, calls the appropriate BatteryMini*Xxx* routine to perform the requested operation.

If **BatteryClassIoctl** returns STATUS_NOT_SUPPORTED for the IRP, the miniclass driver must either complete the IRP or forward it to the next-lower driver.

See Also

BatteryMiniQueryInformation, **BatteryMiniQueryStatus**, **BatteryMiniQueryTag**, **BatteryMiniSetInformation**

BatteryClassStatusNotify

```
NTSTATUS
  BatteryClassStatusNotify (
    IN PVOID ClassData
    );
```

BatteryClassStatusNotify notifies the battery class driver of changes in battery status.

Parameters

ClassData

Points to a battery class handle previously returned by **BatteryClassInitializeDevice**.

Return Value

BatteryClassStatusNotify returns STATUS_SUCCESS.

Comments

Battery miniclass drivers must call **BatteryClassStatusNotify** whenever any of the following occur:

- The battery goes on- or off-line.
- The battery's capacity becomes critically low.
- The battery's power state changes; that is, the battery starts or stops charging or discharging.
- The battery's capacity or power state deviates from the criteria set previously with BatteryMiniSetStatusNotify.

The battery class driver queues status requests internally. If any such requests are pending when the miniclass driver calls **BatteryClassStatusNotify**, the class driver immediately calls the miniclass driver's BatteryMiniQueryStatus routine.

See Also

BatteryMiniQueryStatus, **BatteryMiniSetStatusNotify**

BatteryClassUnload

```
TSTATUS
 BatteryClassUnload (
   IN PVOID ClassData
   );
```

BatteryClassUnload frees resources for a battery device that is no longer in use.

Parameters

ClassData

Points to a battery class handle previously returned by **BatteryClassInitializeDevice**.

Return Value

BatteryClassUnload returns STATUS_SUCCESS.

Comments

BatteryClassUnload frees the battery class handle and unloads the battery device. In essence, it undoes the registration and initialization performed by **BatteryClassInitializeDevice**.

A miniclass driver should call this routine when its battery device is no longer available for use. Typically, the driver might make such a call when handling a PnP IRP_MN_REMOVE_DEVICE request or from its Unload routine.

CHAPTER 4

Battery Miniclass Driver Routines

Battery miniclass drivers must include routines to support PnP and to support battery management and monitoring. Entry points for the following routines are required to support standard operating system and PnP Manager functionality:

- DriverEntry
- AddDevice
- DispatchDeviceControl
- DispatchPnP
- DispatchPower
- Unload

For details on a battery minidriver's support for any of the above routines, see the *Plug and Play, Power Management, and Setup Design Guide*.

Entry points for the following routines, described in this chapter, are required in all battery miniclass drivers:

- **BatteryMiniDisableStatusNotify**
- **BatteryMiniQueryInformation**
- **BatteryMiniQueryStatus**
- **BatteryMiniQueryTag**
- **BatteryMiniSetInformation**
- **BatteryMiniSetStatusNotify**

The battery class driver calls **BatteryMini***Xxx* routines to get and set information about a specific battery device. Battery miniclass drivers must supply entry points for these routines. The **BatteryMini***Xxx* routines can have any name chosen by the driver writer. Prototypes appear in *batclass.h*.

BatteryMiniDisableStatusNotify

```
NTSTATUS
  BatteryMiniDisableStatusNotify(
    IN PVOID Context
    );
```

BatteryMiniDisableStatusNotify disables status notification for a battery device.

Parameters

Context

Points to the miniclass-driver-allocated context area for the battery device.

Return Value

BatteryMiniDisableStatusNotify returns one of the following:

STATUS_SUCCESS

A battery is currently installed and status notification has been disabled.

STATUS_NO_SUCH_DEVICE

No battery is present.

STATUS_NOT_SUPPORTED

No functionality is provided for this routine.

Comments

The battery class driver calls **BatteryMiniDisableStatusNotify** when it no longer requires notification of battery conditions set in an earlier call to **BatteryMiniSetStatusNotify**.

Miniclass drivers that supply a fully functional **BatteryMiniDisableStatusNotify** routine must also supply a fully functional **BatteryMiniSetStatusNotify** routine, and vice versa.

The battery class driver calls this routine at IRQL PASSIVE_LEVEL.

See Also

BatteryMiniSetStatusNotify, **BatteryClassStatusNotify**

BatteryMiniQueryInformation

```
NTSTATUS
  BatteryMiniQueryInformation (
    IN PVOID Context,
    IN ULONG BatteryTag,
    IN BATTERY_QUERY_INFORMATION_LEVEL Level,
    IN LONG AtRate OPTIONAL,
    OUT PVOID Buffer,
    IN ULONG BufferLength,
    OUT PULONG ReturnedLength
    );
```

BatteryMiniQueryInformation returns information about the given battery device.

Parameters

Context

Points to the miniclass-driver-allocated context area for the battery device.

BatteryTag

Points to a battery tag previously returned by **BatteryMiniQueryTag**.

Level

Specifies the type of battery information to be returned. Possible values are **BatteryInformation**, **BatteryGranularityInformation**, **BatteryTemperature**, **Battery-EstimatedTime**, **BatteryDeviceName**, **BatteryManufactureDate**, **BatteryManufacture-Name**, and **BatteryUniqueID**.

AtRate

Specifies the rate of drain, in negative milliwatts, used to calculate the time to discharge the battery. Meaningful only when *Level* is **BatteryEstimatedTime**; ignored for all other values of *Level*.

Buffer

Points to a buffer allocated by the miniclass driver in which to return the requested information. Miniclass drivers format the contents of the buffer depending upon the value of *Level*, as follows:

BatteryInformation

Return information formatted as a BATTERY_INFORMATION structure.

BatteryGranularityInformation

Return a variable-length array of type BATTERY_REPORTING_SCALE that contains the reporting granularity of the remaining capacity. The number of entries returned depends upon the size of the returned buffer, to a maximum of four entries per battery.

BatteryTemperature

Return a ULONG value giving the current temperature of the battery, in tenths of a degree Kelvin.

BatteryEstimatedTime

Return a ULONG value estimating the number of seconds of runtime remaining on the battery, based on the rate of drain specified in *AtRate*. If *AtRate* is negative or zero, the miniclass driver should calculate the runtime based on the current rate of drain. However, if the driver cannot make an estimate (for example, *AtRate* is zero and the battery is not discharging), it should return BATTERY_UNKNOWN_TIME.

BatteryDeviceName

Return a UNICODE string specifying the name of the battery. For example, DR202 identifies a Duracell smart battery.

BatteryManufactureDate

Return a BATTERY_MANUFACTURE_DATE structure specifying the date the battery was manufactured.

BatteryManufactureName

Return a UNICODE string specifying the model name given to the battery by its manufacturer.

BatteryUniqueID

Return a UNICODE string that uniquely identifies the battery, typically a concatenation of the battery's manufacturer, date, and serial number.

BatterySerialNumber

Return a UNICODE string that contains the battery's serial number.

BufferLength

Specifies the length in bytes at *Buffer*.

ReturnedLength

Specifies the number of bytes returned at *Buffer*.

Return Value

BatteryMiniQueryInformation returns one of the following:

STATUS_SUCCESS

The battery designated by *BatteryTag* is currently installed and the requested information has been returned.

STATUS_NO_SUCH_DEVICE

The battery designated by *BatteryTag* is not present.

STATUS_INVALID_DEVICE_REQUEST

The *Level* parameter specifies information that this battery does not support.

STATUS_INVALID_PARAMETER

The *Level* parameter is not one of the enumerators listed.

Comments

The battery class driver calls a miniclass driver's **BatteryMiniQueryInformation** routine to get various types of information about the battery. The information returned depends upon the *Level* parameter. Not all batteries support all the possible types of information that the class driver might request. Miniclass drivers should return STATUS_INVALID_DEVICE_REQUEST for any such requests.

If *Level* specifies **BatteryInformation**, the miniclass driver must return a BATTERY_INFORMATION structure at *Buffer*. This structure contains status information about the battery, including its capabilities, technology (whether the battery is rechargeable), and chemistry; theoretical and actual full-charged capacity; critical bias; number of charge/discharge cycles; and the capacity levels at which warning alerts occur.

If *Level* specifies **BatteryGranularityInformation**, the miniclass driver can return an array of one to four elements, formatted as BATTERY_REPORTING_SCALE structures. Each element of the array consists of a granularity value and a remaining capacity value, in milliwatt-hours. The granularity indicates the precision of measurement and thus is an indicator of the accuracy of the capacity.

Most types of batteries report capacity on a single scale. Miniclass drivers for these batteries return only one entry, giving the remaining capacity and the precision of the scale. Some batteries, however, have two scales: a gross scale that measures whether capacity is greater or less than fifty percent, and a finer scale that applies as capacity approaches zero. Miniclass drivers for such batteries should return two entries describing the two scales.

If *Level* specifies **BatteryEstimatedTime**, the miniclass driver must use the optional *AtRate* parameter to estimate the amount of time remaining to use the battery. The *AtRate* parameter specifies a drain rate, in negative milliwatts.

If *Level* specifies **BatteryUniqueId**, the miniclass driver must return a string that uniquely identifies this particular battery. For control method and smart batteries, the unique ID is the concatenation of the manufacture name, the device name, the manufacture date, and the ASCII representation of the battery's serial number. This value is not meant to be displayed.

The battery class driver calls this routine at IRQL PASSIVE_LEVEL.

See Also

BATTERY_INFORMATION, BATTERY_MANUFACTURE_DATE, BATTERY_REPORTING_SCALE

BatteryMiniQueryStatus

```
NTSTATUS
  BatteryMiniQueryStatus(
    IN PVOID Context,
    IN ULONG BatteryTag,
    OUT PBATTERY_STATUS BatteryStatus
    );
```

BatteryMiniQueryStatus returns status information about the given battery device.

Parameters

Context

Points to the miniclass-driver-allocated context area for the battery device.

BatteryTag

Specifies a battery tag previously returned by **BatteryMiniQueryTag**.

BatteryStatus

Points to a BATTERY_STATUS structure in which the miniclass driver returns information. The BATTERY_STATUS structure is defined as follows:

```
typedef struct _BATTERY_STATUS {
    ULONG    PowerState;
    ULONG    Capacity;
    ULONG    Voltage;
    LONG     Rate;
} BATTERY_STATUS, *PBATTERY_STATUS;
```

PowerState

Specifies a battery power state as one or more of the following flags, ORed together: BATTERY_POWER_ON_LINE, BATTERY_DISCHARGING, BATTERY_CHARGING, and BATTERY_CRITICAL.

Capacity

Specifies the capacity of the given battery, in milliwatt-hours, or BATTERY_UNKNOWN_CAPACITY if the capacity cannot be determined.

Voltage

Specifies the voltage, in millivolts, across the terminals of the given battery, or BATTERY_UNKNOWN_VOLTAGE if the voltage cannot be determined.

Rate

Specifies the current rate of battery usage in milliwatts or, if the driver reports relative capacity, in units per hour. A positive value means that the battery is charging; a negative value means the battery is discharging. If the driver cannot determine the rate, it should return BATTERY_UNKNOWN_RATE.

Return Value

BatteryMiniQueryStatus returns one of the following:

STATUS_SUCCESS

The battery designated by *BatteryTag* is currently installed.

STATUS_NO_SUCH_DEVICE

The battery designated by *BatteryTag* is not present.

Comments

The battery class driver calls **BatteryMiniQueryStatus** to get status information about the battery. The status information includes the battery's power state, capacity, voltage, and the amount of current flowing at the time of the request.

If the miniclass driver does not supply fully functional **BatteryMiniSetStatusNotify** and **BatteryMiniDisableStatusNotify** routines, the battery class driver calls **BatteryMini-QueryStatus** at regular but infrequent intervals to poll the battery's status. Otherwise, the class driver calls this routine after the miniclass driver has notified it of a change in battery status.

Before reporting a critically low, discharging battery (BATTERY_DISCHARGING and BATTERY_CRITICAL), the miniclass driver should ensure that the problem is legitimate (rather than a transitory state) and if so, should attempt to solve the problem. Possible solutions might include switching to AC power or to another battery. When the miniclass driver reports that a battery is critical and discharging, the system assumes that battery failure is imminent and prepares to shut down.

The battery class driver calls this routine at IRQL PASSIVE_LEVEL.

See Also

BatteryClassStatusNotify, **BatteryMiniDisableStatusNotify**, **BatteryMiniSetStatus-Notify**

BatteryMiniQueryTag

```
NTSTATUS
  BatteryMiniQueryTag(
    IN PVOID Context,
    OUT PULONG BatteryTag
    );
```

BatteryMiniQueryTag returns the current battery tag.

Parameters

Context

Points to the miniclass-driver-allocated context area for the battery device.

BatteryTag

Points to a caller-allocated variable in which the miniclass driver returns the battery tag.

Return Value

BatteryMiniQueryTag returns one of the following:

STATUS_SUCCESS

A battery is currently installed.

STATUS_NO_SUCH_DEVICE

No battery is present.

Comments

The battery class driver calls **BatteryMiniQueryTag** to get the value of the current battery tag. If a battery is present, **BatteryMiniQueryTag** should return the tag in *BatteryTag* and return STATUS_SUCCESS.

Each time a battery is inserted, the miniclass driver must increment the value of the tag, regardless of whether this is a new battery or the same battery that was previously present.

If no battery is present, or if the miniclass driver cannot determine whether a battery is present, it should return STATUS_NO_SUCH_DEVICE and set the value of the tag to BATTERY_TAG_INVALID.

The battery class driver calls this routine at IRQL PASSIVE_LEVEL.

BatteryMiniSetInformation

```
NTSTATUS
  BatteryMiniSetInformation(
    IN PVOID Context,
    IN ULONG BatteryTag,
    IN BATTERY_SET_INFORMATION_LEVEL Level,
    IN PVOID Buffer OPTIONAL
    );
```

BatteryMiniSetInformation requests that a battery enter the charging or discharging state, or sets a critical bias value for the battery.

Parameters

Context

Points to the miniclass-driver-allocated context area for the battery device.

BatteryTag

Specifies a battery tag previously returned by **BatteryMiniQueryTag**.

Level

Specifies one of the following values: **BatteryCriticalBias**, **BatteryCharge**, or **BatteryDischarge**.

Buffer

Specifies the critical bias adjustment in milliwatts if *Level* is **BatteryCriticalBias**. Not used for other values of *Level.*

Return Value

BatteryMiniSetInformation returns one of the following:

STATUS_SUCCESS

The operation succeeded.

STATUS_NO_SUCH_DEVICE

No battery is present.

STATUS_NOT_SUPPORTED

The specified battery does not support the requested operation.

STATUS_UNSUCCESSFUL

The operation failed.

Comments

The battery class driver calls **BatteryMiniSetInformation** to request that a battery start to charge or discharge. It can also call this routine to set a critical bias value.

With a smart battery charger/selector, the class driver specifies **BatteryCharge** to select a battery to charge, possibly discontinuing the charging of another battery.

The class driver specifies **BatteryDischarge** to indicate which battery should power the system.

The critical bias adjustment is analogous to the reserve capacity of the gas tank in an automobile. It represents the remaining charge when the battery capacity is reported as zero. Although the class driver does not change the critical bias value in normal use, this field is provided in the interface as a maintenance feature. Not all batteries can maintain a critical bias setting. Miniclass drivers for such batteries should return STATUS_NOT_SUPPORTED.

The battery class driver calls this routine at IRQL PASSIVE_LEVEL.

BatteryMiniSetStatusNotify

```
NTSTATUS
  BatteryMiniSetStatusNotify(
    IN PVOID Context,
    IN ULONG BatteryTag,
    IN PBATTERY_NOTIFY BatteryNotify
    );
```

BatteryMiniSetStatusNotify sets battery capacity and power state levels at which the class driver requires notification.

Parameters

Context

Points to the miniclass-driver-allocated context area for the battery device.

BatteryTag

Specifies a battery tag previously returned by **BatteryMiniQueryTag**.

BatteryNotify

Points to a BATTERY_NOTIFY structure, defined as follows:

```
typedef struct {
    ULONG    PowerState;
    ULONG    LowCapacity;
    ULONG    HighCapacity;
} BATTERY_NOTIFY, *PBATTERY_NOTIFY;
```

PowerState

Sets one or more of the following flags to specify a battery power state: BATTERY_POWER_ON_LINE, BATTERY_DISCHARGING, BATTERY_CHARGING, BATTERY_CRITICAL.

LowCapacity

Specifies a ULONG value indicating the battery capacity below which the class driver requires notification.

HighCapacity

Specifies a ULONG value indicating the battery capacity above which the class driver requires notification.

Return Value

BatteryMiniSetStatusNotify returns one of the following:

STATUS_SUCCESS

A battery is currently installed.

STATUS_NO_SUCH_DEVICE

No battery is present or the given battery tag is invalid.

STATUS_NOT_SUPPORTED

The miniclass driver cannot distinguish the target condition.

Comments

The battery class driver calls a miniclass driver's **BatteryMiniSetStatusNotify** routine to set criteria for an acceptable range of battery conditions. When the battery's capacity or power state deviates from these criteria, the miniclass driver must call **BatteryClassStatusNotify** to notify the class driver.

In **PowerState**, the class driver specifies one or more battery power states. Any time the battery enters a power state that is not in **PowerState**, the miniclass driver must notify the class driver.

In **LowCapacity** and **HighCapacity**, the class driver specifies a range of capacity. When the capacity falls above or below this range, the miniclass driver must notify the class driver.

Some batteries might be unable to distinguish the precise capacities requested by the battery class driver. When possible, miniclass drivers for these batteries should attempt to correct for the error so that the user can be informed when the battery approaches a critical state. Otherwise, such drivers should return STATUS_NOT_SUPPORTED.

The battery class driver calls this routine at IRQL PASSIVE_LEVEL.

See Also

BatteryClassStatusNotify, **BatteryMiniDisableStatusNotify**

CHAPTER 5

Battery Structures

This chapter describes the following structures used by battery miniclass drivers:

- BATTERY_INFORMATION
- BATTERY_MANUFACTURE_DATE
- BATTERY_REPORTING_SCALE

BATTERY_INFORMATION

```
typedef struct _BATTERY_INFORMATION {
  ULONG    Capabilities;
  UCHAR    Technology;
  UCHAR    Reserved[3];
  UCHAR    Chemistry[4];
  ULONG    DesignedCapacity;
  ULONG    FullChargedCapacity;
  ULONG    DefaultAlert1;
  ULONG    DefaultAlert2;
  ULONG    CriticalBias;
  ULONG    CycleCount;
} BATTERY_INFORMATION, *PBATTERY_INFORMATION;
```

Battery miniclass drivers fill in this structure in response to certain **BatteryMiniQuery-Information** requests.

Members

Capabilities

Specify battery capabilities as a ULONG value encoded with one or more of the following flags:

BATTERY_SYSTEM_BATTERY

Set this flag if the battery can provide general power to run the system.

BATTERY_CAPACITY_RELATIVE

Set this flag if the miniclass driver will report battery capacity and rate as a percentage of total capacity and rate rather than as absolute values. Otherwise, the miniclass driver should report capacity in milliwatt-hours and rate in milliwatts.

BATTERY_IS_SHORT_TERM

Set this flag if the battery is a UPS, intended for short-term, failsafe use. Clear the flag for any other type of device.

BATTERY_SET_CHARGE_SUPPORTED

Set this flag if the miniclass driver supports the **BatteryCharge** setting in calls to **BatteryMiniSetInformation**.

BATTERY_SET_DISCHARGE_SUPPORTED

Set this flag if the miniclass driver supports the **BatteryDischarge** setting in calls to **BatteryMiniSetInformation**.

Technology

Specify zero for a primary, nonrechargeable battery, or one for a secondary, rechargeable battery.

Chemistry

Specify a four-character string indicating the type of chemistry used in the battery. Possible values include PbAc (Lead Acid), LION (Lithium Ion), NiCd (Nickel Cadmium), NiMH (Nickel Metal Hydride), NiZn (Nickel Zinc), and RAM (Rechargeable Alkaline-Manganese). Additional values might be returned as additional battery types become available.

DesignedCapacity

Specify the theoretical capacity of the battery when new, in milliwatt-hours. If BATTERY_ CAPACITY_RELATIVE is set, the units are undefined.

FullChargedCapacity

Specify the battery's current fully charged capacity, in milliwatt-hours. If BATTERY_ CAPACITY_RELATIVE is set, the units are undefined.

DefaultAlert1

Specify the capacity (in milliwatt-hours) at which a low battery alert should occur.

DefaultAlert2

Specify the capacity (in milliwatt-hours) at which a warning battery alert should occur.

CriticalBias

Specify the amount (in milliwatt-hours) of any small reserved charge remaining when the critical battery level shows zero. Miniclass drivers should subtract this value from the battery's **FullChargedCapacity** and remaining capacity (reported in BATTERY_STATUS) before reporting those values.

CycleCount

Specify the number of charge/discharge cycles the battery has experienced, or zero if the battery does not support a cycle counter.

See Also

BatteryMiniQueryInformation, **BatteryMiniQueryStatus**

BATTERY_MANUFACTURE_DATE

```
typedef struct _BATTERY_MANUFACTURE_DATE {
  UCHAR     Day;
  UCHAR     Month;
  USHORT    Year;
} BATTERY_MANUFACTURE_DATE, *PBATTERY_MANUFACTURE_DATE;
```

Battery miniclass drivers fill in this structure in response to certain **BatteryMiniQuery-Information** requests.

Members

Day

Specify a value in the range 1 to 31, inclusive.

Month

Specify a value in the range 1 to 12, inclusive.

Year

Specify a value >= 1996.

See Also

BatteryMiniQueryInformation

BATTERY_REPORTING_SCALE

```
typedef struct {
  ULONG    Granularity;
  ULONG    Capacity;
} BATTERY_REPORTING_SCALE;
```

Battery miniclass drivers fill in this structure in response to certain **BatteryMiniQuery-Information** requests.

Members

Granularity

Specify the granularity of the **Capacity** value, in milliwatt-hours. For most batteries, this value describes a monotonically increasing scale of capacity. For lithium ion batteries, this value describes one of two possible scales: a gross measure of battery capacity, with a large granularity, or a finer measure as the capacity approaches zero.

Capacity

Specify the battery capacity described by the corresponding granularity, in milliwatt-hours.

See Also

BatteryMiniQueryInformation

PART 3

Setup

C H A P T E R 1

INF File Sections and Directives

This chapter describes the syntax of INF files. See *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide* for additional information on creating and using INF files.

This introduction contains the following information:

- *General Syntax Rules for INF Files*
- *Looking at an INF File*
- *Summary of INF Sections*
- *Summary of INF Directives*

General Syntax Rules for INF Files

An INF file is a simple text file organized into named sections.

Some sections have system-defined names and some sections have names determined by the writer of the INF. Each section contains section-specific entries and/or directives that reference additional sections specified elsewhere in the INF file. Each section, section-specific entry, and directive has a particular purpose, for example, to copy files from the distribution media, to install a driver service, or to add (or modify) the value entries in registry keys.

The rest of this discussion describes syntax rules governing the required contents of INF files, the format of section names, using string tokens, and line format, continuation, and comments.

Required Contents

- The set of required and optional sections, entries, and directives in any particular INF depends on the type of device/driver or component (such as an application or device class installer DLL) to be installed. The set of sections, section-specific entries, and directives required to install any particular device and its driver(s) also depends somewhat on the corresponding class installer. For more information about how the system-supplied class installers handle device-type-specific INF files, see also the *Plug and Play, Power Management, and Setup Design Guide* and, for certain types of devices, the appropriate manual in this documentation set.

- Sections can be specified in any order. Most INFs list sections in a particular order, by convention, but Setup finds sections by name, not by location within the INF file.

Section Names

- Each section in an INF begins with the section name enclosed in brackets ([ ]). The section name can be system-defined or INF-writer-defined.

  For example, **[Manufacturer]** specifies the start of the system-named **Manufacturer** section, while [Std.Mfg] represents a particular INF-writer-defined *Models* section name.

  A section name has a maximum length of 255 bytes on Microsoft® Windows® 2000. On Windows 98, section names should be no longer than 28 characters. INFs designed to work on both platforms must adhere to the smaller limit.

  Each section ends at the beginning of a new [*section-name*] or at the end-of-file mark.

- If more than one section in an INF has the same name, the system merges their entries and directives into a single section.

- Section names, entries, and directives are case-insensitive, so **version**, **VERSION**, and **Version** are equally valid section-name specifications within an INF.

- Unless it is enclosed in double-quote characters ("), an INF-writer-defined section name referenced elsewhere in the INF file must be a unique-to-the-INF unquoted string of explicitly visible characters, excluding certain characters with INF-specific meanings. In particular, an unquoted section name referenced by a section entry or directive cannot have leading or trailing spaces, a linefeed character, a return character, or any invisible control character, and it should not contain tabs. In addition, it cannot contain either of the bracket ([ ]) characters, a single percent (%) character, a semicolon (;), or any internal double-quote (") character(s), and it cannot have a backslash (\) as its last character.

  For example, Std.Mfg and Std_Mfg are unique and valid section names when referenced by n INF entry or directive, but Std;Mfg (with its internal semicolon) is invalid unless it is enclosed by double quotes (").

  However, specifying an INF-writer-defined section name as a "*quoted string*" overrides most of the the preceding restrictions on characters in referenced section names. Such a delimited section name can contain almost any explicitly or implicitly visible characters except the closing bracket (]) as long as the corresponding section in the INF matches this "*quoted string*" exactly.

For example, ";;   Std Mfg   " is a valid section-name reference if the corresponding section declaration in the INF exactly matches the name inside the double quotes with respect to its space and semicolon characters as [;;   Std Mfg   ].

Using String Tokens

- Many values in an INF, including INF-writer-defined section names, can be expressed as tokens of the form *%strkey%*. Each such *strkey* must be defined in the **Strings** section of the INF file as a value consisting of a sequence of explicitly visible characters, which the Windows 2000 setup code converts, if necessary, into Unicode internally. (See the reference for the **Strings** section for more detailed information about how to define *%strkey%* tokens and their respective values.)

Line Format, Continuation, and Comments

- Each entry and directive in a section ends with a return or linefeed character. Consequently, the text editor used to create an INF file must not insert return or linefeed characters after some arbitrary, editor-determined number of characters.

- The backslash character (\) can be used as an explicit line continuator in an entry or directive. If part of an entry or directive, such as a path, includes a backslash at the end of a line, that backslash must be delimited with double quotes ("\") to override its interpretation as a line continuator.

- Comments begin with a semicolon (;) character. When parsing and interpreting an INF file, the system assumes that the following have no relevance to the installation process:

  (1) Any characters following a semicolon on the same line, unless the semicolon appears within a "*quoted string*" or *%strkey%* token

  (2) Any empty line containing nothing except a linefeed or return character

- Commas separate the values supplied in section entries and directives.

  An INF entry or directive can omit an optional value in the middle of a list of values, but the commas must remain. Windows 2000 INFs can omit trailing commas, but Windows 9x INFs must not. Dual-OS INFs should specify trailing commas in any sections that are used on Windows 9x machines. Dual-OS INFs can omit trailing commas in sections that are only used in Windows 2000 (that is, sections whose names are decorated with **.nt**, **.ntx86**, and so forth).

For example, consider the syntax for a **SourceDisksFiles** section entry:

filename = diskid[,[subdir][,size]]

An entry that omits the *subdir* value but supplies the *size* value must specify both delimiting internal commas, as follows:

filename = diskid,,size

An entry in a Windows 2000 INF that omits the two optional values can have this format:

filename = diskid

An entry in a Windows9x INF that omits the two optional values must specify the trailing commas, as follows:

filename = diskid,,

Looking at an INF File

The following example shows selected fragments from a system-supplied class installer's INF file to illustrate how any INF file is made up of sections, each containing zero or more lines, some of which are entries that reference additional INF-writer-defined sections:

```
[Version]
Signature="$Windows NT$"
Class=Mouse
ClassGUID={4D36E96F-E325-11CE-BFC1-08002BE10318}
Provider=%Provider% ; defined later in Strings section
LayoutFile=layout.inf ; entry used only by system installers
DriverVer=09/28/1999,5.00.2136.1

; ... some class installer sections omitted here

[DestinationDirs]
DefaultDestDir=12 ; DIRID_DRIVERS

; ... [ControlFlags] section omitted here

[Manufacturer]
%StdMfg%    =StdMfg        ; (Standard types)
%MSMfg%     =MSMfg         ; Microsoft
; ... %otherMfg% entries omitted here

[StdMfg]  ; per-Manufacturer Models section
; Std serial mouse
%*pnp0f0c.DeviceDesc%= Ser_Inst,*PNP0F0C,SERENUM\PNP0F0C,SERIAL_MOUSE
; Std InPort mouse
```

```
%*pnp0f0d.DeviceDesc%      = Inp_Inst,*PNP0F0D
     ; ... more StdMfg entries and following
     ; MSMfg and xxMfg Models sections omitted here

     ; per-Models DDInstall (Ser_Inst, Inp_Inst, etc.)
     ; sections also omitted here

[Strings]
; where INF %strkey% tokens are defined as user-visible (and
; possibly as locale-specific) strings.
Provider = "Microsoft"
; ...
StdMfg   = "(Standard mouse types)"
MSMfg    = "Microsoft"

; ...
*pnp0f0c.DeviceDesc   = "Standard Serial Mouse"
*pnp0f0d.DeviceDesc   = "InPort Adapter Mouse"
; ...
HID\Vid_045E&Pid_0009.DeviceDesc = "Microsoft USB Intellimouse"
; ...
```

A few sections within the preceding Windows 2000 INF have system-defined names, such as **Version**, **DestinationDirs**, **Manufacturer**, and **Strings**. Some named sections like **Version**, **DestinationDirs**, and **Strings** have only simple entries. Others reference additional INF-writer-defined sections, as shown in the preceding example of the **Manufacturer** section.

Note the implied hierarchy of related sections for mouse device driver installation(s) starting with the **Manufacturer** section in the preceding example. Figure 1.1 on the next page shows the hierarchy of some of the sections in the INF file.

Note the following about the implied hierachy of an INF file:

- Each *%xx*Mfg*%* entry in the **Manufacturer** section references a per-manufacturer *Models* section (StdMfg, MSMfg) elsewhere in the INF. (The entries in the example above use *%strkey%* tokens.)

- Each *Models* section specifies some number of entries; in the example they are *%xxx*.DeviceDesc*%* tokens.

  Each such *%xxx*.DeviceDesc*%* token references some number of per-model(s) *DDInstall* sections (Ser_Inst and Inp_Inst) for that manufacturer's product line, with each entry identifying a single device (*PNP0F0C and *PNP0F0D, hence the "DeviceDesc" shown here) or a set of compatible models of a device.

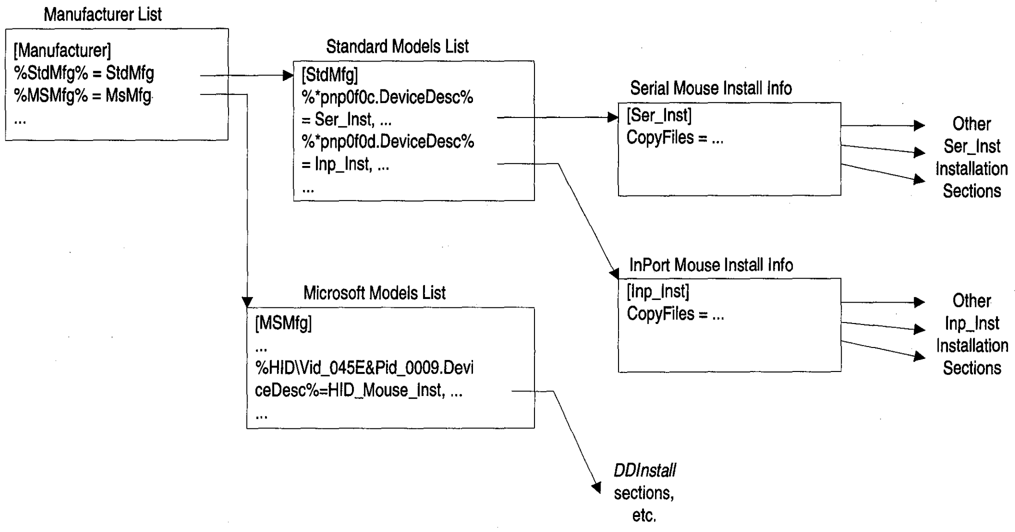


Figure 1.1 Sample Hierarchy of Sections in an INF File

- Each such *DDInstall*-type *Xxx*_Inst section, in turn, can have certain system-defined extensions appended and/or can contain directives that reference additional INF-writer-defined sections. For instance, the full INF shown as fragments in the preceding example also has a Ser_Inst.**Services** section, and its Ser_Inst section has a **CopyFiles** directive that references a Ser_CopyFiles section elsewhere in this INF.

Summary of INF Sections

The following summarizes the system-defined sections that can be used in INF files. System-defined section names are case-insensitive, so **version**, **VERSION**, and **Version** are equally valid section-names within an INF.

This chapter describes the INF sections in the same general order used in most device INF files. However, these sections actually can be specified in any arbitrary order. The Windows 2000 setup code finds all sections within each INF file by section name, not by sequential order, whether system-defined or INF-writer-defined.

Version Section

This is a required section for every INF file. For installation on Windows 2000 and/or Windows 9x platforms, this section must have a valid **Signature** entry.

SourceDisksNames Section

This section is required if the INF has a corresponding **SourceDisksFiles** section. This section is required to install IHV/OEM-supplied devices and their drivers from distribution media included in packaged products. It is also required in such an INF that installs either of the following:

A coinstaller DLL to supplement the operations of a system-supplied device class installer or coinstaller(s) (see also *DDInstall*.**CoInstallers** later in this list)

A new class installer DLL to supplement the operations of the OS's device installer (see also **ClassInstall32** later in this list)

This section identifies the individual source distribution disks or CD-ROM discs for the installation. By contrast, the system-supplied INFs each specify a **LayoutFile** entry in their **Version** sections and provide at least one other INF file detailing the source distribution contents and layout of all software components to be installed.

SourceDisksFiles Section

This section identifies the location(s) of files to be installed from the distribution media to the destination(s) on the target machine. An INF that has this section must also have a **SourceDisksNames** section.

ClassInstall32 Section

This section initializes a device setup class. This section is required in any class installer INF file, but see also *DDInstall*.**CoInstallers** later in this list. INFs that install devices and their drivers under any system-defined device class do not need this section.

DestinationDirs Section

Device/driver INFs have a **DestinationDirs** section to specify a default destination directory for INF-specified copies of the files supplied on the distribution media or listed in the INF layout file(s). This section is required unless the INF installs a device, such as a modem or display monitor, that has no files except its INF to be installed with it.

ControlFlags Section

This section controls whether the Add New Hardware Wizard presents a list of INF-specified *Models* values from which the end user selects a particular manually installed device (or model of a device) to be installed from the INF. It also can control whether an INF is used only to transfer files from the distribution media.

In general, most INFs for device drivers and for the system class installers have this section so they can exclude at least a subset of *Models* entries from the list of manually installable devices to be displayed to end users. INFs that only install PnP devices suppress the display of all model-specific information.

Manufacturer Section

This section is required in INFs for devices and their drivers.

The **Manufacturer** section of a system device class INF is sometimes called a "Table of Contents," because each of its entries references an INF-writer-defined *Models* section, which, in turn, references additional INF-writer-defined sections, such as a per-models-entry *DInstall* section, *DDInstall*.**Services** section, and so forth.

Models Section (per Manufacturer entry)

This section is required to identify the device(s) for which the INF installs driver(s). It specifies a set of mappings between the generic name (string) for a device, the device ID, and the name of the *DDInstall* section, elsewhere in the INF, containing the installation instructions for the device.

An INF that installs one or more devices and driver(s) for a single provider would have only one *Models* section, but system INFs for device classes can have many INF-writer-defined *Models* sections.

DDInstall Section (per Models entry)

This section is required to actually install any device(s) listed in a *Models* section in the INF, along with the driver(s) for each such device. A *DDInstall* section can be shared by more than one *Models* section.

DDInstall.Services Section

This section is required as an expansion of the *DDInstall* section for most Windows 2000 kernel-mode device drivers, including any WDM drivers (exceptions are INFs for modems and display monitors). It controls how and when the services of a particular driver are started, its dependencies (if any) on underlying legacy drivers, and so forth. This section also sets up event-logging services by a device driver if it supports event logging.

DDInstall.HW Section

This optional section adds device-specific (and typically, driver-independent) information to the registry or removes such information from the registry, possibly for a multifunction device or to install one or more PnP filter drivers.

DDInstall.CoInstallers Section

This optional section registers one or more device-specific or class-specific coinstallers supplied on the distribution media to supplement the operations of the system's device intaller, of an existing device class installer, and/or of existing class-specific coinstaller(s), if any.

A device-specific coinstaller is an IHV/OEM-provided Win32® DLL that typically writes additional configuration information to the registry or performs other device installation tasks that require dynamically generated, machine-specific information that is not available when the device's INF is created. A class-specific coinstaller is also a Win32 DLL that supplements the installation operations of an already installed device class installer or of the system's device installer.

DDInstall.Interfaces Section

If a device/driver "exports" certain system-defined device interfaces, such as kernel-streaming still-image capture or data decompression, or it exports a new class of device interface to higher level components, its INF can have this section.

InterfaceInstall32 Section

If a to-be-installed component, such as a new class driver, provides one or more new device interfaces to higher-level components, its INF has this section. In effect, this section bootstraps a set of device interface(s) of a new class by setting up whatever is needed to make use of the functionality that interface class provides.

DDInstall.FactDef Section

This section should be included in the INF of any manually installed nonPnP device. It specifies the factory default hardware configuration settings, such as the bus-relative I/O ports, IRQ (if any), and so forth, for the card.

Strings Section

This section is required in every INF file to define each *%strkey%* token specified in the INF. By convention, the **Strings** section (or sections if the INF provides a set of locale-specific **Strings** sections) appear(s) last in all system-supplied INF files for ease of maintenance and localization.

Some of the sections listed here, particularly those with *Install* in their names, can contain directives that reference additional INF-writer-defined sections. Each directive causes particular operations to be carried out on the items listed under the appropriate type of INF-writer-defined section during the installation process.

The set of valid entries and/or directives for any particular section in the preceding list is section-specific and shown in the formal syntax of the reference for each of these sections. Optional entries and directives within each such section are shown enclosed in unbolded brackets, as for example:

[Version]

...

[**Provider=***%INF-creator%*]

...

The **Provider** entry in a **[Version]** section is optional in the sense that it is not a mandatory entry in every INF file.

Summary of INF Directives

The following summarizes the system-defined directives that can be used in INF files. INF directive names are case-insensitive, so **Addreg**, **addReg**, and **AddReg** are equally valid as directive specifications within an INF.

This chapter lists the most commonly used directives first, together with their reciprocal or related directives. The most rarely used directives are toward the end of the chapter.

AddReg Directive

This directive references one or more *add-registry-section*s used to add subkeys and/or value entries to the registry or to modify existing value entries.

The particular INF section in which an **AddReg** (or **DelReg**) directive occurs determines the default ("relative") location within the registry for the modifications specified in the referenced add-registry (or delete-registry) section. For device/driver INFs, these default registry locations, which can be designated in the INF's add/delete-registry sections by the value **HKR**, are typically user-visible, device-specific or driver-specific subkeys somewhere under the following keys in the HKEY_LOCAL_MACHINE registry tree:

- **..Enum**, under which the system's PnP enumerators store device-specific information, such as the device ID, compatible device IDs, if any, and so forth

  Any INF-supplied information stored somewhere in the **..Enum** (sometimes called the "device" or "hardware") branch of the registry is generally device-specific but driver-independent in nature. For example, an add-registry section referenced in the *DDInstall*.**HW** INF section can be used to write the value entries of a **Device Parameters** subkey under the device-specific key in the **..Enum** branch. The OS creates such a device-specific subkey of the **..Enum** branch for each detected and enumerated PnP device and for legacy (nonPnP) devices. As another example, device-specific logical-configuration information, whether supplied by a PnP bus driver or by an *add-registry-section* referenced in an INF-writer-created *log-config-section* is also stored in a subkey of such a device-specific key.

  In the Windows 2000 registry, this **..Enum** branch is a subtree of the **..\CurrentControlSet\Control** tree.

- **..Class\***SetupClassGUID*, in which the system's device installer stores information about each particular (setup) class of devices and under which the corresponding device class installer and coinstallers, if any, store per-device/driver information, such as the "friendly name" of a particular device, the device description string, the name of the device's manufacturer, the name of the driver image, and so forth

  For example, any add-registry section referenced in an INF's *DDInstall* section is assumed to store this kind of registry information, possibly some of it as localized string values, in a subkey (sometimes called the "software" or "driver" key) under the **..Class\***SetupClassGUID* key for the appropriate device class.

In the Windows 2000 registry, this **..Class** branch is also a subtree of the **..\CurrentControlSet\Control** tree.

Additional INF sections referenced by **AddReg** can set up registry information for supplied coinstallers, for system-defined device interfaces (such as kernel streaming interfaces) exported to higher level components by a device/driver, for new device interfaces exported by an installed component for a given class of devices, for driver services, and/or even for a new setup class of devices if the INF has a **ClassInstall32** section.

DelReg Directive

This directive references one or more *del-registry-section*s used to remove obsolete subkeys and/or value entries from the registry. For example, such a section might appear in an INF that upgrades a previous installation.

CopyFiles Directive

This directive references one or more *file-list-section*s specifying transfers of model/device-specific driver image(s) and any other necessary files from the distribution media to the destination directory for each such file. Alternatively, this directive can specify a single file to be copied from the distribution media to the default destination directory.

DelFiles Directive

This rarely used directive references one or more *file-list-section*s specifying files to be deleted from the target of the installation. For example, such an optional section might appear in an INF to "deinstall" file(s) that will be superceded by some file(s) to be installed by the INF. If no such stale file(s) could possibly be installed on the target machine, this directive is irrelevant.

RenFiles Directive

This rarely used directive references one or more *file-list-section*s specifying INF-associated source files to be renamed on the destination. For example, such an optional section might appear in an INF if the installer should change the name(s) of one or more "replaced" files on the destination to preserve them when copying files supplied on the distribution media.

AddService Directive

This directive references at least a *service-install-section*, possibly with an additional *event-log-install-section*.

INFs for most kinds of Windows 2000 devices (those that install drivers) have an INF-writer-defined *service-install-section* to specify any dependencies on system-supplied drivers or services, during which stage of the system initialization process the supplied driver(s) should be loaded, and so forth. Many INFs for device drivers also have an

INF-writer-defined *event-log-install-section* that is referenced by the **AddService** directive to set up event logging by the device driver.

DelService Directive

This rarely used directive deletes a previously installed service. For example, it might undo the operations of an **AddService** directive specified in a previous version of the INF file.

AddInterface Directive

This directive references an *add-interface-section* in which one or more **AddReg** directives are specified referencing sections that set up the registry entries for the device interfaces supported by this device/driver. Optionally, such an *add-interface-section* can reference one or more additional sections that specify delete-registry, file-transfer, file-delete, and/or file-rename operations.

BitReg Directive

This rarely used directive references one or more *bit-registry-section*s specifying existing REG_BINARY-type value entries in the registry for which particular bits in the values are to be modified.

LogConfig Directive

This directive references one or more *log-config-section*s that specify acceptable bus-relative and device-specific hardware configurations in an INF for device(s) that are detected (by PnP device enumerators) or manually installed. For example, INFs for nonPnP ISA, EISA, and MCA devices, which are manually installed, use this directive. (See also this directive's reference for more information about the even more rarely used *DDInstall*.**LogConfigOverride** section.)

UpdateInis Directive

This rarely used directive references one or more *update-ini-section*s specifying parts of a supplied INI file to be read during installation and, possibly specifying line-by-line modifications to be made in that INI file.

UpdateIniFields Directive

This rarely used directive references one or more *update-inifields-section*s specifying modifications to be made on fields within the lines of an INI file.

Ini2Reg Directive

This rarely used directive references one or more *ini-to-registry-section*s specifying lines or sections of an INI file to be written into the registry.

The specific set of valid sections under which any of the directives in the preceding list can be be specified is system-determined. For quick reference, the basic form of each valid section is shown later in the formal syntax of the reference for each directive, as for example:

[*DDInstall*] | [*DDInstall*.**HW**] | [*DDInstall*.**CoInstallers**] | [**ClassInstall32**] |
[**ClassInstall32.ntx86**]

AddReg=*add-registry-section*[**,** *add-registry-section*] ...

However, the system-defined extensions for cross-platform Windows 2000 and dual-OS INF files can be appended to certain INF-writer-defined section names, as explained in *Creating an INF File*. That is, the undecorated [*DDInstall*.**HW**] section shown in the formal syntax for the **AddReg** directive reference implies the validity of all decorated forms of this type of section, such as [*install-section-name*.**nt.HW**], [*install-section-name*.**ntx86.HW**], and so forth.

The rest of this chapter describes the formal syntax and meaning for each system-defined named section, standard INF-writer-defined section, and directive that can be specified in an INF file.

INF Version Section

[Version]

Signature="*signature-name***"**
[**Class=***class-name*]
[**ClassGuid={***nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn***}**]
[**Provider=***%INF-creator%*]
[**LayoutFile=***filename***.inf** [**,***filename***.inf**]...]
[**CatalogFile=***filename***.cat**]
[**CatalogFile.nt=***unique-filename***.cat**]
[**CatalogFile.ntx86=***unique-filename***.cat**]
DriverVer=*mm/dd/yyyy*[**,***x.y.v.z*]

By convention, the **Version** section appears first in INF files. Every INF file must have this section.

Entries and Values

Signature="signature-name"

Can be any of $Windows NT$, $Chicago$, or $Windows 95$. The enclosing $ are required but these strings are otherwise case-insensitive. If the *signature-name* is none of these string values, the file is not accepted as a valid INF.

If an INF is used to install device(s)/driver(s) on both Windows 9x and Windows 2000 platforms, it must designate any OS-specific installation information by appending system-defined extension(s) to its *DDInstall* section(s), whether the *signature-name* is $Windows NT$, $Chicago$, or $Windows 95$. (See *Creating an INF File* in the *Plug and Play, Power Management, and Setup Design Guide* for a discussion of these extensions.)

Class=class-name

For any standard type of device, this specifies the class name, such as one of the system-defined class names like **Net** or **Display** as listed in *devguid.h*, for the type of device to be installed from this INF file. See *Device Setup Classes* for more information on the system-defined device setup classes.

If an INF specifies a **Class** it should also specify the corresponding system-defined GUID value for its **ClassGUID** entry. Specifying the matching GUID value for a device of any predefined device setup class can install such a device and its driver(s) faster since this helps the system setup code to optimize its INF searching.

Any INF that adds a new setup class of devices to the system should supply a unique, case-insensitive *class-name* value that is different from any of the system-defined device-type-specific classes in *devguid.h*. Such an INF must specify a newly generated GUID value for the **ClassGUID** entry. Otherwise, this entry is irrelevant to an INF that installs neither a new device driver under a predefined device setup class nor a new device setup class.

ClassGuid = {nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}

Specifies the device-class GUID, formatted as shown here, where each *n* is a hexadecimal digit.

For a Windows 2000 device/driver(s) INF, such a GUID value determines the device (setup) class subkey in the registry **...\Class** tree under which to write registry information for the driver(s) of device(s) installed from this INF file. This class-specific GUID value also identifies the device-class installer for the type of device and class-specific property-page provider, if any.

For a new device setup class, the INF must specify a newly generated **ClassGUID** value. For more information about creating GUIDs, see the *Plug and Play, Power Management, and Setup Design Guide*. For more information about system-defined device setup classes, see *Device Setup Classes*.

Provider=%INF-creator%

Identifies the provider of the INF file. Typically, this is specified as an *%Organization-Name%* token that is expanded later in the INF's **Strings** section.

For example, INFs supplied with the system typically specify the *INF-creator* as %Msft% and define %Msft% = "Microsoft" in their **Strings** sections.

LayoutFile=filename.inf [,filename.inf]...

Specifies one or more additional system-supplied INF files that contain layout information on the source media required for installing the software described in this INF. All system-supplied INFs specify this entry, but IHV/OEM-supplied INFs do not.

INF files that are not distributed with the OS itself must omit this entry and have **Source-DisksNames** and **SourceDisksFiles** sections instead. By convention, these two sections follow the **Version** section.

CatalogFile=filename.cat

Specifies the catalog file to be included on the distribution media of a device/driver when it has been tested, certified, and assigned digital signature(s) by the Microsoft Windows Hardware Quality Lab. (Contact WHQL for more information about the signing, testing, and certification of IHV and/or OEM driver packages.) The file has the extension **.cat**.

Catalog files are not listed in the **SourceDisksFiles** or **CopyFiles** sections of the INF. Setup assumes that the catalog file is in the same location as the INF file.

System-supplied INF files never have **CatalogFile=** entries because the OS validates the signature for such an INF against all system-supplied *xxx*.**cat** files.

[CatalogFile.nt=unique-filename.cat] | [CatalogFile.ntx86=unique-filename.cat] |

Specifies another INF-writer-determined but unique file name, also with the extension **.cat**, for a Windows 2000-specific or Windows 2000-platform-specific catalog file to be included on the distribution media of a device/driver already validated by WHQL.

If these optional entries are omitted from a dual-OS INF file, a given **CatalogFile=** *filename*.**cat** is used for validating WDM device/driver installations on all Windows 2000 and Windows 98 machines. If any decorated **CatalogFile.***xxx***=** entry exists in an INF's **Version** section together with an undecorated **CatalogFile=** entry, the undecorated entry is assumed to identify a *filename*.**cat** for validating device/driver installations only on Windows 98 machines.

Note that any cross-platform and/or dual-OS device/driver INF file that has **CatalogFile=** and **CatalogFile.***xxx***=** entries must supply a unique IHV/OEM-determined name for each such **.cat** file.

DriverVer=mm/dd/yyyy[,x.y.v.z]

This entry specifies version information for drivers installed by this INF. This entry is required in Windows 2000 INFs.

The *mm/dd/yyyy* value specifies the date of the driver package, including the driver files and the INF. A hyphen (-) can be used as the date field separator in place of the slash (/).

The *x.y.v.z* specifies an optional version number. This value is for display purposes only (for example, in the Device Manager). The OS does not use this value for driver selection.

A Windows 2000 INF should have a **DriverVer** entry in the **Version** section to provide version information for the whole INF plus **DriverVer** directives in the individual *DDInstall* sections to provide version information for individual drivers. **DriverVer** entries in the

DDInstall sections are more specific and take precedence over the global **DriverVer** entry in the **Version** section.

When the OS searches for drivers, it chooses a driver with a more recent **DriverVer** date over a driver with an earlier date. If an INF has no **DriverVer** entry or is unsigned, the OS applies the default date of 00/00/0000.

Windows 98 does not recognize a **DriverVer** entry in the **Version** section. Therefore, an INF that will be used on Windows 98 should have **DriverVer** entries in the undecorated *DDInstall* sections that are used by Windows 98.

Comments

When the Microsoft Windows Hardware Quality Lab certifies a driver package, it returns **.cat** catalog file(s) to the IHV or OEM that contain the digitally encrypted signature(s) for the driver package. The IHV or OEM must list any **.cat** file(s) in the **Version** section of their INF and must supply the files on the distribution media in the same location as the IHV/OEM-supplied INF file. The **.cat** files must be in uncompressed form.

Example

The following example shows a **Version** section typical of a simple device-driver INF, followed by the required **SourceDisksNames** and **SourceDisksFiles** sections implied by the entries specified in this sample **Version** section:

```
[Version]
Signature="$Chicago$"
Class=SCSIAdapter
ClassGUID={4D36E97B-E325-11CE-BFC1-08002BE10318}
Provider=%INF_Provider%
CatalogFile=aha154_win98.cat
CatalogFile.ntx86=aha154_ntx86.cat
DriverVer=08/20/1999

[SourceDisksNames]
;
; diskid = description[, [tagfile] [, <unused>, subdir]]
;
1 = %Floppy_Description%,,,\Win98
2 = %Floppy_Description%,,,\WinNT

[SourceDisksFiles]
;
; filename_on_source = diskID[, [subdir][, size]]
;
aha154x.mpd = 1,,
[SourceDisksFiles.x86]
aha154x.sys = 2,\x86

; ...

[Strings]
INF_Provider="Adaptec"
Floppy_Description = "Adaptec Drivers Disk"
; ...
```

See Also

DDInstall, **SourceDisksNames**, **SourceDisksFiles**, **Strings**

INF SourceDisksNames Section

[SourceDisksNames] |
[SourceDisksNames.x86]

diskid = *%strkey%* | ["]*disk-description*["][,[*tagfile*][,*unused*,*path*]]
...

A **SourceDisksNames** section identifies the distribution disk(s) or CD-ROM disc(s) that contain the source files to be transferred to the target machine during installation.

Entry Values

diskid

Specifies a nonnegative integer that identifies a source disk. This value can be expressed in decimal or in hexadecimal notation, but it cannot require more than four bytes of storage. If there is more than one source disk for the distribution, each *diskid* entry in this section must have a unique value, such as **1**, **2**, **3**,... or **0x0**, **0x1**, **0x2**,... and so forth.

disk-description

Specifies a *%strkey%* token or a "*quoted string*" that describes the contents and/or purpose of the disk identified by *diskid*. The installer can display the value of this string to the end user during installation, for example, to identify a source disk to be inserted into a drive at a particular stage of the installation process.

Every *%strkey%* specification in this section must be defined in the INF's **Strings** section. Any *disk-description* that is not a *%strkey%* token is a user-visible string that must be delimited by double-quote characters (") if it has any leading or trailing spaces.

tagfile

This optional value specifies the name of a tag file supplied on the distribution disk, either in the root directory or in the given *path* subdirectory, if any, of the disk. The value should specify only the filename, not any directory or subdirectory.

Setup uses a tag file to verify that the user inserted the correct installation disk. Tag files are only used for removeable media.

A vendor can also use a tag file to contain a "cabinet" of compressed installation files. If *tagfile* has the extension **.cab**, Setup uses it as a tag file *and* as a source of installation files.

unused

This is not used on Windows 2000. This value is only used in Windows 9x. See the Windows 98 DDK documentation for further information.

path

This optional value specifies the path to the directory on the distribution disk containing source files, including the *tagfile* if any. The *path* is relative to the root and is expressed as *\dirname1\dirname2*... and so forth. If this value is omitted from an entry, files are assumed to be in the root directory of the distribution disk.

Subdirectories containing particular source files can be specified relative to a given *path* directory in the corresponding **SourceDisksFiles** section of the INF file. However, any *tagfile* supplied on the distribution disk must reside either in the given *path* directory or, if *path* is omitted, in the root directory.

Comments

A **SourceDisksNames** section can have any number of entries, one for each distribution disk. Any INF with a **SourceDisksNames** section also must have a **SourceDisksFiles** section.

These sections never appear in system-supplied INFs. Instead, system-supplied INFs specify **LayoutFile** entries in their **Version** sections.

To support a multiplatform distribution of Windows 2000 driver files, construct platform-specific **SourceDisksNames** sections. For example, all system setup API functions that process a **SourceDisksNames** section will search first for a **SourceDisksNames.x86** section on a Windows 2000 x86-based platform and only look at an undecorated **SourceDisksNames** section if they cannot find a **SourceDisksNames.x86** section.

Examples

In the following example, the *write.exe* file is the same for all Windows 2000 platforms and is located in the *\common* directory on a CD-ROM distribution disc. The *cmd.exe* file is a platform-specific file that is only used on Windows 2000 x86-based platforms.

```
[SourceDisksNames]
1 = "Windows NT CD-ROM",file.tag,,\common

[SourceDisksNames.x86]
2 = "Windows NT CD-ROM",file.tag,,\x86

[SourceDisksFiles]
write.exe = 1
cmd.exe = 2
```

This next example again shows the **SourceDisksNames** section from the example in the preceding reference for the **Version** section.

```
[SourceDisksNames]
;
; diskid = description[, [tagfile] [, <unused>, subdir]]
;
1 = %Floppy_Description%,,,\Win98
2 = %Floppy_Description%,,,\WinNT
```

See Also

DestinationDirs, **SourceDisksFiles**, **Version**

INF SourceDisksFiles Section

[SourceDisksFiles] |
[SourceDisksFiles.x86]

filename = *diskid*[,[*subdir*][, *size*]]
...

A **SourceDisksFiles** section names the source files used during installation, identifies the source disks (or CD-ROM discs) that contain those files, and provides the path to the subdirectories, if any, on the distribution disks containing individual files.

Entry Values

filename

Specifies the name of the file on the source disk.

diskid

Specifies the integer identifying the source disk that contains the file. This value and the initial *path* to the *subdir*(ectory), if any, containing the named file must be defined in a **SourceDisksNames** section of the same INF.

subdir

This optional value specifies the subdirectory (relative to the **SourceDisksNames** *path* specification, if any) on the source disk where the named file resides.

If this value is omitted from an entry, the named source file is assumed to be in the root directory or in the *path* directory that was specified in the **SourceDisksNames** section for the given disk.

size

This optional value specifies the uncompressed size, in bytes, of the given file.

Comments

A **SourceDisksFiles** section can have any number of entries, one for each file on the distribution disk(s). Any INF with a **SourceDisksFiles** section also must have a **SourceDisksNames** section. (These sections are omitted from a system-supplied INF, which instead specifies a **LayoutFile** entry in its **Version** section.)

To support a multiplatform distribution of Windows 2000 source files, construct platform-specific **SourceDisksFiles** sections. For example, all system setup API functions that process a **SourceDisksFiles** section will search first for a **SourceDisksFiles.x86** section on a Windows 2000 x86-based platform and only look in an undecorated **SourceDisksFiles** section if they cannot find a **SourceDisksFiles.x86** section.

However, the presence of a **SourceDisksFiles.x86** section does not exclude the existence of an undecorated **SourceDisksFiles** section within the same INF if it installs some software that is cross-platform in nature.

Example

The following example shows the **SourceDisksFiles** section for the corresponding **SourceDisksNames** example shown in the immediately preceding section and in the **Version** section.

```
[SourceDisksFiles]
;
; filename_on_source = diskID[, [subdir][, size]]
;
aha154x.mpd = 1,, ; on distribution disk 1, in subdir \win9x

[SourceDisksFiles.x86]
aha154x.sys = 2,\x86 ; on distribution disk 2, in subdir \WinNT\x86

; ...
```

See Also

CopyFiles, **DestinationDirs**, **RenFiles**, **SourceDisksNames**, **Version**

INF ClassInstall32 Section

[ClassInstall32] | [ClassInstall32.ntx86]

AddReg=*add-registry-section*[**,** *add-registry-section*] ...
[**Copyfiles=@***filename* | *file-list-section*[**,** *file-list-section*] ...]
[**DelReg=***del-registry-section*[**,** *del-registry-section*] ...]
[**Delfiles=***file-list section*[**,** *file-list-section*] ...]
[**Renfiles=***file-list-section*[**,** *file-list-section*] ...]
[**BitReg=***bit-registry-section*[**,***bit-registry-section*]...]
[**UpdateInis=***update-ini-section*[**,***update-ini-section*]...]
[**UpdateIniFields=***update-inifields-section*[**,***update-inifields-section*]...]
[**Ini2Reg=***ini-to-registry-section*[**,***ini-to-registry-section*]...]

A **ClassInstall32** section installs a new setup device class (and possibly a class installer) for some number of devices of the same new type.

An INF for device(s) in a system-defined device setup class should not specify a **ClassInstall32** section. However, coinstaller(s) can be provided for any such device or class of devices to supplement the installation operations of existing class installers or of the Windows 2000 device installer.

Usually, a **ClassInstall32** section will have one or more **AddReg** directives to add value entries under a system-provided *SetupClassGUID* subkey in the registry. These value entries can include the class-specific "friendly name," class-installer path specification, class icon, property-page provider, if any, and so forth. Except for **AddReg** and **CopyFiles**, the other directives shown here are very seldom used in a **ClassInstall32** section.

Valid Directives

AddReg=add-registry-section[, add-registry-section] ...

References one or more named sections in which class-specific value entries are specified to be written into the registry when this INF is processed. Typically, this is used to give the new device setup class at least a friendly name that other components can later retrieve from the registry and use to open installed devices of this new device class, to "install" any new device class installer and/or property-page provider for this device setup class, and so forth. An **HKR** specification in any *add-registry-section* referenced here designates the **..Class\\{***SetupClassGUID***}** registry key.

Copyfiles=@filename | file-list-section[, file-list-section] ...

Either specifies one named file to be copied from the source media to the destination or references one or more named sections in which class-relevant file(s) on the source media are specified for transfer to the destination. The **DefaultDestDir** entry in the **DestinationDirs** section of the INF specifies the destination directory for any class-specific single file to be copied.

System-supplied INFs for device setup classes (and class installers) do not use this directive in this section.

DelReg=del-registry-section[, del-registry-section] ...

References one or more named sections in which value entries or keys are specified to be removed from the registry during installation of the class installer.

However, if a particular {*SetupClassGUID*} subkey exists in the registry **..Class** branch, the system setup code subsequently ignores the **ClassInstall32** section of any INF that specifies the same GUID value in its **Version** section. Consequently, an INF cannot replace an existing class installer or modify its behavior from a **ClassInstall32** section. To modify the behavior of existing class installer, use a class-specific coinstaller.

Delfiles=file-list section[, file-list-section] ...

References one or more named sections in which previously installed class-relevant file(s) on the destination are specified for deletion.

Renfiles=file-list-section[, file-list-section] ...

References one or more named sections in which class-relevant file(s) to be renamed on the destination are listed.

BitReg=bit-registry-section[,bit-registry-section]...

Is valid in this section but almost never used.

UpdateInis=update-ini-section[,update-ini-section]...

Is valid in this section but almost never used.

UpdateIniFields=update-inifields-section[,update-inifields-section]...

Is valid in this section but almost never used.

Ini2Reg=ini-to-registry-section[,ini-to-registry-section]...

Is valid in this section but almost never used.

Comments

The system processes the **ClassInstall32** section of an INF for a new device setup class when such a device is about to be installed but the *SetupClassGUID* value of that device's class is not predefined by Windows 2000. (See *Device Setup Classes* for a list of the system-defined setup class names and GUIDs.)

To support a multiplatform distribution of Windows 2000 driver files, construct platform-specific **ClassInstall32** sections. For example, all system setup API functions that process a **ClassInstall32** section will search first for a **ClassInstall32.ntx86** section on a Windows 2000 x86-based platform and only look at an undecorated **ClassInstall32** section if they cannot find a **ClassInstall32.ntx86** section.

Every device installed on Windows 2000 platforms is associated with a device setup class in the registry. If the INF for a particular device to be installed is not associated with a new device class installer or its **ClassGUID=** specification in the **Version** section does not match any of the system-defined setup class GUIDs, that device's registry subkey is created under **..Class\\{***UnknownClassGUID***}**.

The INF for any device class installer typically has an **AddReg** directive in its **ClassInstall32** section to define at least one named section that creates a friendly name for its kind of device in the *SetupClassGUID* subkey of the registry **..Class** tree. The Windows 2000 setup code automatically creates this *SetupClassGUID* subkey in the registry from the value supplied for the **ClassGUID=** entry in the **Version** section of such an INF file when the first device of that (new) setup class is installed.

Under this *SetupClassGUID* subkey, such an INF also provides registry information for some number of *Models*-specific subkeys, using additional **AddReg** directives in its per-manufacturer, per-models *DDInstall* sections. In addition, the INF can use the add-registry section(s) referenced in its **ClassInstall32** section to specify a property-page provider and to exert control over how its class of devices is handled in the user interface.

Such a class-specific add-registry section has the following general form to define a friendly name for the device setup class and other class-specific value entries in the class-specific registry key:

[*SetupClassAddReg*]

HKR,,,,*%DevClassName%* ; device-class friendly name
[**HKR,,Installer32,,**"*class-installer***.dll,***class-entry-point*"]
[**HKR,,EnumPropPages32,,**"*prop-provider***.dll,***provider-entry-point*"]
HKR,,Icon,,"*icon-number*"
[**HKR,,SilentInstall,,1**]
[**HKR,,NoInstallClass,,1**]
[**HKR,,NoDisplayClass,,1**]

A nonnegative *icon-number* indicates the icon is supplied by the given property-page provider or by the new device class installer. Negative *icon-number* values are reserved for system use.

Setting the predefined **SilentInstall**, **NoDisplayClass**, and **NoInstallClass** Boolean value entries in a class-specific registry key has the following effects:

- Setting **SilentInstall** directs installers to send no pop-ups to the user that require a response while installing device(s) of this class, whether specified in the *DDInstall* sections of the class installer's INF file or in separate INF files for subsequently installed devices that declare themselves of this class by setting the same **ClassGuid={***Class-GUID***}** specification in their respective **Version** sections. For example, the system class installers of CD-ROM and disk devices and the system parallel port class installer set **SilentInstall** in their respective registry keys.

  If a class-specific installer requires the machine to be rebooted for any device that it installs, the class-specific add-registry section in its INF cannot have this value entry.

- Setting **NoDisplayClass** suppresses the user-visible display of all devices of this class by the Device Manager. For example, the system class installers for printers and for network drivers (including clients, services, and protocols) set **NoDisplayClass** in their respective registry keys.

- Setting **NoInstallClass** indicates that no device of this type will ever require manual installation by an end user. For example, the system class installers for exclusively PnP devices set **NoInstallClass** in their respective registry keys.

A **ClassInstall32** section can contain **AddReg** directives to set the **DeviceType**, **Device-Characteristics**, and **Security** for devices of its setup class. See the *INF AddReg Directive* for more information.

Examples

This example shows the **ClassInstall32** section, along with the named section it references with the **AddReg** directive, of the INF for the system display class installer.

```
[ClassInstall32]
AddReg=display_class_addreg

[display_class_addreg]
HKR,,,,%DisplayClassName%
HKR,,Installer32,,"Desk.Cpl,DisplayClassInstaller"
HKR,,Icon,,"-1"
```

By contrast, this example shows the add-registry section referenced in the system CD-ROM INF's **ClassInstall32** section. It sets up a class-specific property-page provider for the CD-ROM devices/drivers that it installs. This INF also sets the **SilentInstall** and **NoInstallClass** value entries in the CD-ROM class key to TRUE (**1**).

```
[cdrom_class_addreg]
HKR,,,,%CDClassName%
HKR,,EnumPropPages32,,"SysSetup.Dll,CdromPropPageProvider"
HKR,,SilentInstall,,1
HKR,,NoInstallClass,,1
HKR,,Icon,,"101"
```

See Also

AddReg, **BitReg**, **CopyFiles**, ***DDInstall***, **DelFiles**, **DelReg**, **Ini2Reg**, ***Models***, **RenFiles**, **SetupDiBuildClassInfoListEx**, **UpdateIniFields**, **UpdateInis**, **Version**

INF DestinationDirs Section

[DestinationDirs]

[**DefaultDestDir=***dirid*[,*subdir*]]
[*file-list-section*=*dirid*[,*subdir*]] ...

A **DestinationDirs** section specifies the target destination directory or directories for all copy, delete, and/or rename operations on files referenced by name elsewhere in the INF file.

This section is required in any INF that uses a **CopyFiles** directive or that references a *file-list-section*, whether with a **CopyFiles**, **DelFiles**, or **RenFiles** directive.

Entry Values

DefaultDestDir=dirid[,subdir]]

Specifies the default destination directory for all copy, delete, and/or rename operations on files that are not explicitly listed in a *file-list-section* referenced by other entries here.

file-list-section

Specifies the INF-writer-determined name of a section referenced by a **CopyFiles**, **RenFiles**, or **DelFiles** directive elsewhere in the INF file. Such an entry is optional if this section has a **DefaultDestDir** entry and all copy-file operations specified in this INF have the same target destination. However, any *file-list-section* referenced by a **RenFiles** or **DelFiles** directive elsewhere in the INF must be listed here.

dirid

Specifies the directory identifier of the target directory for operations on files that are referenced by name, possibly within a named *file-list-section* of the INF. This can be one of the following numerical values, which are shown here with the values most commonly specified by device/driver INFs toward the top of this list:

Value	Destination Directory
12	Drivers directory This is equivalent to *%windir%\system32\drivers* on Windows 2000 platforms and to *%windir%\system\IoSubsys* on Windows 9x platforms.
10	Windows directory This is equivalent to *%windir%* for both Windows 2000 and Windows 9x.
11	System directory This is equivalent to *%windir%\system32* for Windows 2000 and to *%windir%\system* for Windows 9x.
50	*%windir%\system* directory (Windows 2000 only)
30	Root directory of the boot drive (a.k.a. "ARC system partition" for Windows 2000)
54	Directory where *ntldr.exe* and *osloader.exe* are located (Windows 2000 only)
01	*SourceDrive*:*pathname* (the directory from which the INF file was installed)
17	INF file directory
20	Fonts directory
51	Spool directory
52	Spool drivers directory
55	Print processors directory
23	Color (ICM)
-1	Absolute path
21	Viewers directory
53	User Profile directory
24	Applications directory
25	Shared directory
18	Help directory

The following *dirid* values are for commonly used shell "special folders":

Value	Shell Special Folder
16406	All Users\Start Menu
16407	All Users\Start Menu\Programs
16408	All Users\Start Menu\Programs\Startup
16409	All Users\Desktop
16415	All Users\Favorites
16419	All Users\Application Data
16422	Program Files

Value	Shell Special Folder
16427	Program Files\Common
16429	All Users\Templates
16430	All Users\Documents

Besides the values listed above that are defined in *setupapi.h*, you can use any of the CSIDL_*Xxx* values defined in *shlobj.h*. To define a *dirid* value for a folder not listed above, add 16384 (0x4000) to the CSIDL_*Xxx* value. For more information on CSIDL_*Xxx* values, see the Platform SDK documentation.

subdir

Specifies the subdirectory (and the rest of its path, if any, under the directory identified by *dirid*) to be the destination of the file operations in the given *file-list-section*.

Comments

The optional **DefaultDestDir** entry provides a default destination for copy, rename, and delete file operations that appear elsewhere in the INF file:

- **CopyFiles** directives that use the direct copy (*@filename*) notation must have a **Default-DestDir** entry in the **DestinationDirs** section of the INF in which the direct-copy entry appears.
- **CopyFiles**, **RenFiles**, or **DelFiles** sections that are not directly referenced in the **DestinationDirs** section must have a **DefaultDestDir** entry in the **DestinationDirs** section of the INF in which the copy/rename/delete files section(s) appear.

Because all WDM drivers must be installed in the *%windir%\system32\drivers* directory of computers running Windows 2000 or Windows 98, their dual-OS INFs must specify the *dirid* value **10** with an explicit *subdir* path as **system32\drivers** either as the **DefaultDest-Dir** entry, if any, or in the given *file-list-section*(s) (referenced elsewhere in the INF with the **CopyFiles** directive) that list the WDM driver images to be copied to the target.

Examples

This example sets the default target directory for all copy-file, delete-file, and rename-file operations specified in a given Windows 2000 INF file to the *%windir%***\system32**\drivers directory. Such a simple **DestinationDirs** section is common to INFs for new Windows 2000 peripheral devices, because such an INF usually just copies a set of source files into a single directory on the target machine.

```
[DestinationDirs]
DefaultDestDir = 12 ; dirid = \Drivers on WinNT platforms
```

This example shows a fragment of the **DestinationDirs** section of the INF for Windows 2000-installed display/video drivers, for which the **ClassInstall32** section was shown as an example in the immediately preceding reference.

```
[DestinationDirs]
DefaultDestDir     = 11 ; dirid = \system32 on WinNT platforms

; ...

; list of per-Manufacturer, per-Models, per-DDInstall-section, and
; CopyFiles-referenced xxx.Miniport/xxx.Display sections omitted here
; along with several other miniport/display paired drivers
; ...
vga.Miniport     = 12
vga.Display      = 11
xga.Miniport     = 12
xga.Display      = 11

; all video miniports copied into \system32\drivers on WinNT platforms
; all paired display drivers copied into \system32
```

See Also

ClassInstall32, **CopyFiles**, ***DDInstall***, **DelFiles**, **RenFiles**, **SourceDisksFiles**, **SourceDisksNames**, **Version**

INF ControlFlags Section

[ControlFlags]

ExcludeFromSelect=* |
ExcludeFromSelect=*hw-id*[,*hw-id*] ...
[**ExcludeFromSelect.nt=***hw-id*[,*hw-id*] ...]
[**ExcludeFromSelect.ntx86=***hw-id*[,*hw-id*] ...]
[**CopyFilesOnly=***hw-id*[,*hw-id*] ...]
[**InteractiveInstall=***hw-id*[,*hw-id*] ...]

Typically, a **ControlFlags** section has one or more **ExcludeFromSelect** entries to control which device(s) listed in the per-manufacturer *Models* section of INF file will not be displayed to the end user as options during manual installations.

INFs that install exclusively PnP devices also have this section unless they set the **NoInstallClass** value entry in their respective *SetupClassGUID* keys to TRUE, as already described in the reference for the **ClassInstall32** section.

Entries and Values

ExcludeFromSelect

Removes all (*) or the specified list of devices from the display shown to the end user, from which that user is expected to select a particular device for installation. To exclude a set of OS-incompatible or platform-incompatible devices from this display, one or more **Exclude-FromSelect** entries can have the following system-defined (and case-insensitive) extensions appended:

.nt

Do not display these device(s) on computers running Windows 2000.

.ntx86

Do not display these device(s) on x86-based computers running Windows 2000.

hw-id

Identifies a device that is specified in the per-manufacturer *Models* section of the INF file. Each such *hw-id* value in a given entry must be separated from the next with a comma (,).

CopyFilesOnly

Installs only the INF-specified files for the given device(s) because the device hardware is not accessible or available yet. This entry is rarely used. However, it can be used to pre-install the driver(s) of a device for which the card will later be seated in a particular slot that is currently in use. For example, if a device currently seated in the particular slot is necessary to transfer INF-specified files to the target, the INF would have this entry.

InteractiveInstall

Forces the specified list of devices to be installed in a user's context. Each line can specify one or more hardware or compatible IDs and there can be one or more lines.

This entry is optional. The preferred way to install devices is to omit this entry and allow Setup to install the device in the context of a trusted system thread, if possible. However, if a device absolutely requires a user to be logged in when the device is installed, include this entry in the device INF. This entry is not supported on Windows 9x systems.

Comments

The system's New Device Wizard builds a list of installable devices by searching through all available INF files. It extracts information about models/devices from each of these INF files and displays this information to the end user, unless an INF overrides this behavior by suppressing the display of one or more models/devices in that INF's **ControlFlags** section.

Listing the *hw-id* of a device in an **ExcludeFromSelect** entry removes it from the display shown to the end user. Specifying * (an asterisk) for the **ExcludeFromSelect** value removes all devices/models defined in the INF file from this user-visible list.

An INF writer should use the **InteractiveInstall** sparingly and only in the following situations:

- To install driver(s) for devices that have corrupted or otherwise incorrectly defined hardware IDs. For example, when two or more different devices share the same Hardware ID. This case is strictly forbidden by the Plug and Play standard, but some hardware vendors have made this error in hardware design.
- To install drivers for devices that require their own driver and absolutely cannot use the generic class driver or another driver supplied with Windows 2000. The **InteractiveInstall** directive forces the Windows 2000 Device Manager to ask the user for confirmation for compatible ID matches.

In the future, WHQL might not grant the Windows Logo to devices whose INF files include any **InteractiveInstall** entries.

Example

This example of the **ControlFlags** section in the system mouse class installer INF suppresses the display of devices/models that cannot be installed on Windows 2000 x86-based platforms. (Some relevant fragments of the same INF were already shown in the introduction to this chapter.)

```
[ControlFlags]
; Exclude all bus mice and InPort mice for x86 platforms
ExcludeFromSelect.ntx86=*PNP0F0D,*PNP0F11,*PNP0F00,*PNP0F02,*PNP0F15
; Hide this entry always
ExcludeFromSelect=UNKNOWN_MOUSE
```

The following INF file fragment shows two devices: one that is fully PnP capable and requires no user intervention during installation and another that requires its own driver and cannot use any other driver. Specifying **InteractiveInstall** for the second device forces Windows 2000 to install this device in a user's context (a user with administrative rights), including prompting the user for the location of the driver files (INF file, driver file, and so on) as required.

```
; ...
[Manufacturer]
%Mfg% = ModelsSection

[ModelsSection]
; Models section, with two entries
%Device1.DeviceDesc% = Device1.Install, \
  PCI\VEN_1000&DEV_0001&SUBSYS_00000000&REV_01
%Device2.Device.Desc% = Device2.Install, \
  PCI\VEN_1000&DEV_0001&SUBSYS_00000000&REV_02
```

```
[ControlFlags]
InteractiveInstall = \
  PCI\VEN_1000&DEV_0001&SUBSYS_00000000&REV_02
; ...
```

See Also

ClassInstall32, **Manufacturer**, ***Models***

INF Manufacturer Section

[Manufacturer]

manufacturer-name
[*manufacturer-name*] ... |
%strkey%=models-section-name
[*%strkey%=models-section-name*] ...

The **Manufacturer** section identifies the manufacturer of one or more devices that can be installed using the INF file. It also defines the Models section name for the installation of that manufacturer's devices and their driver(s).

Entries and Values

manufacturer-name

Identifies the device(s)' manufacturer and the corresponding *Models* section elsewhere in the INF. Each such entry must uniquely identify the manufacturer within the INF file. However, an entry specified in this manner cannot be localized.

strkey

Specifies a token, unique within the INF, representing the name of a manufacturer. Each such *%strkey%* token must be defined in a **Strings** section of the INF file.

models-section-name

Specifies an INF-writer-defined name for the per-manufacturer *Models* section within the INF file. This value must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*.

Comments

Any INF that installs one or more devices must have a **Manufacturer** section. An IHV/OEM-supplied INF typically specifies only a single entry in this section, but using a *%strkey%=models-section-name* entry simplifies the localization of the INF for the international market, as described in *Creating International INF Files* or as described later in the **Strings** section.

If an INF file specifies one or more entries in the *manufacturer-name* format, each such entry implicitly specifies the name of the corresponding *Models* section elsewhere in the INF.

The **Manufacturer** section of a system-supplied device setup class INF is sometimes called a "Table of Contents" because this section sets up the installation of every manufacturer's devices/models of the same class for which the drivers are supplied with the OS. Each entry in such an INF's **Manufacturer** section specifies both an easily localizable *%strkey%* token for the name of a manufacturer and a unique-to-the-INF per-manufacturer *Models* section name.

Examples

This example shows a **Manufacturer** section typical to an INF for a single IHV.

```
[Manufacturer]
%LogiMfg%=LogiMfg        ; Models section == LogiMfg

; ...
[Strings]
LogiMfg = "Logitech"
```

The next example shows part of a **Manufacturer** section typical to an INF for a device-class-specific installer:

```
[Manufacturer]
%ADP%=ADAPTEC
; several entries omitted here for brevity
%SONY%=SONY
%ULTRASTOR%=ULTRASTORE
```

See Also

Models, **Strings**

INF Models Section

[*models-section-name*]

device-description=*install-section-name*,*hw-id*[,*compatible-id*...]
[*device-description*=*install-section-name*,*hw-id*[,*compatible-id*]...] ...

A per-manufacturer *Models* section identifies at least one device, references the *DDInstall* section of the INF file for that device, and specifies a unique-to-the-INF hardware identifier for that device. Any entry in the per-manufacturer *Models* section also can specify one or more additional device ID(s) for model(s) compatible with the device designated by the initial HW ID and controlled by the same driver(s).

Each INF-writer-defined *models-section-name* must be referenced in the **Manufacturer** section of the INF file. There can be one or more entries in any per-manufacturer *Models* section, depending upon how many devices (and drivers) the INF file installs for a particular manufacturer.

Entry Values

device-description

Identifies a device to be installed, expressed as any unique combination of explicitly visible characters or as a *%strkey%* token defined in a **Strings** section of the INF file.

install-section-name

Specifies an INF-writer-determined name of the *DDInstall* section for the device (and compatible models of device, if any).

hw-id

Specifies a vendor-defined string that identifies a device, which the PnP Manager uses to find an INF-file match for this device. Such a hardware ID has one of the following formats:

enumerator\enumerator-specific-device-id

Is the typical format for individual PnP devices reported to the PnP Manager by a single enumerator. For example, USB\VID_045E&PID_00B identifies the Microsoft HID keyboard device on a USB bus. Depending on the enumerator, such a specification can even include the device's hardware revison number as, for example, PCI\VEN_1011&DEV_002&SUBSYS_00000000&REV_02.

*enumerator-specific-device-id

Indicates with the asterisk (*) that the device is supported by more than one enumerator. For example, *PNP0F01 identifies the Microsoft serial mouse, which also has a *compatible-id* specification of SERENUM\PNP0F01.

device-class-specific-ID

Is an I/O bus-specific format, as described in the hardware specification for the bus, for the hardware IDs of all peripheral devices on that type of I/O bus.

Note that a single device can have more than one *hw-id* value. The PnP Manager uses each such *hw-id* value, which is usually provided by the underlying bus when it enumerates its child devices, to create a subkey for each such device in the registry **..Enum** branch. For manually installed devices, the system's setup code uses their *hw-id* values as specified in their respective INF files to create each such registry subkey.

compatible-id

Specifies a *hw-id* value compatible with that designated by the given *hw-id*. Any number of *compatible-id* values can be specified for an entry in the *Models* section, each separated from the next by a comma (,). All such compatible devices and/or device models are controlled by the same driver as the device designated by the initial *hw-id.*

Comments

Any given *install-section-name* must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*. Such a *DDInstall* section name referenced in a per-manufacturer *Models* section also can have extensions appended to the given *install-section-name*, thus defining additional *DDInstall* sections for the OS-specific or platform-specific installation of the given device(s). For more information about using extensions in cross-platform Windows 2000 and/or dual-OS files, see also *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide.*

Any given *hw-id* for a manually installed device can be specified in the **ControlFlags** section of the INF to prevent that device from being displayed to the end user by the Add New Hardware Wizard.

For more information about PnP *hw-id* and *compatible-id* values, see also the *Plug and Play, Power Management, and Setup Design Guide.*

For each device/driver installed using an INF file, the device installer(s) use the information supplied in the **Manufacturer** section and per-manufacturer *Models* sections to generate Device Description, Manufacturer Name, Device ID if the installation is manual, and, possibly, Compatibility List value entries in the registry.

Example

This example shows a per-manufacturer *Models* section with some representative entries from the system mouse class installer's INF file, defining the *DDInstall* sections for some devices/models.

```
[Manufacturer]
%StdMfg%     =StdMfg         ; (Standard types)
%MSMfg%      =MSMfg          ; Microsoft
; ... %otherMfg% omitted here

[StdMfg]  ; per-Manufacturer Models section
  ; Std serial mouse
%*pnp0f0c.DeviceDesc%= Ser_Inst,*PNP0F0C,SERENUM\PNP0F0C,SERIAL_MOUSE
  ; Std InPort mouse
%*pnp0f0d.DeviceDesc%      = Inp_Inst,*PNP0F0D
; ... more StdMfg entries
```

See Also

ControlFlags, ***DDInstall***, **Manufacturer**, **Strings**

INF DDInstall Section

[*install-section-name*] |
[*install-section-name*.**nt**] |
[*install-section-name*.**ntx86**]

[**DriverVer=***mm/dd/yyyy*[,*x.y.v.z*]]
[**CopyFiles=**@*filename* | *file-list-section*[,*file-list-section*] ...]
AddReg=*add-registry-section*[,*add-registry-section*]...
[**Include=***filename.inf*[,*filename2.inf*]...]
[**Needs=***inf-section-name*[,*inf-section-name*]...]
[**Delfiles=***file-list-section*[,*file-list-section*]...]
[**Renfiles=***file-list-section*[,*file-list-section*]...]
[**DelReg=***del-registry-section*[,*del-registry-section*]...]
[**BitReg=***bit-registry-section*[,*bit-registry-section*]...]
[**LogConfig=***log-config-section*[,*log-config-section*]...]
[**ProfileItems=***profile-items-section*[,*profile-items-section*]...]
[**UpdateInis=***update-ini-section*[,*update-ini-section*]...]
[**UpdateIniFields=***update-inifields-section*[,*update-inifields-section*]...]
[**Ini2Reg=***ini-to-registry-section*[,*ini-to-registry-section*]...]
...

Each per-Models *DDInstall* section contains an optional **DriverVer** entry and one or more directives referencing additional named sections in the INF file, shown here with the most commonly specified INF directives, **CopyFiles** and **AddReg**, listed first. The sections referenced by these directives contain instructions for installing driver files and writing any device-specific and/or driver-specific information into the registry.

Directives and Entries

DriverVer=mm/dd/yyyy[,x.y.v.z]

This optional entry specifies version information for the driver package.

The *mm/dd/yyyy* value specifies the date of the driver package, including the driver files and the INF. This date should be the most recent date of any file in the driver package. A hyphen (-) can be used as the date field separator in place of the slash (/).

The *x.y.v.z* specifies an optional version number. This value is for display purposes only (for example, in the Device Manager). The OS does not use this value for driver selection.

When the OS searches for drivers, it chooses a driver with a more recent **DriverVer** date over a driver with an earlier date. If an INF has no **DriverVer** entry or is unsigned, the OS applies the default date of 00/00/0000.

A Windows 2000 INF should have a **DriverVer** entry in its **Version** section and/or in the individual *DDInstall* sections. If there is a **DriverVer** entry in a *DDInstall* section, the OS uses that entry instead of the one in the **Version** section for that particular device.

Windows 98 does not recognize a **DriverVer** entry in the **Version** section. Therefore, an INF that will be used on Windows 98 should have **DriverVer** entries in the undecorated *DDInstall* sections that are used by Windows 98.

CopyFiles=@filename | file-list-section[,file-list-section] ...

This directive either specifies one named file to be copied from the source media to the destination or references one or more INF-writer-defined sections in which device-relevant file(s) on the source media are specified for transfer to the destination. The **CopyFiles** directive is optional, but is present in most *DDInstall* sections.

The **DefaultDestDir** entry in the **DestinationDirs** section of the INF specifies the destination for any single file to be copied. The **SourceDisksNames** and **SourceDisksFiles** sections, or an additional INF specified in the **LayoutFile** entry of this INF's **Version** section, provides the location on the distribution media of the driver file(s).

AddReg=add-registry-section[,add-registry-section]...

This directive references one or more INF-writer-defined sections in which new subkeys, possibly with initial value entries, are specified to be written into the registry or in which the value entries of existing keys are modified.

An **HKR** specification in such an add-registry section designates the **..Class\***SetupClass-GUID\device-instance-id* registry path to the user-accessible driver (a.k.a. "software" key).

Include=filename.inf[,filename2.inf]...

This optional entry specifies one or more additional named INF files containing sections needed to install this device and/or driver. If this entry is specified, usually so is a **Needs** entry.

For example, the system INFs for device drivers that depend on the system's kernel-streaming support specify this entry as **Include= ks.inf[, [kscaptur.inf,] [ksfilter.inf]]**.

Needs=inf-section-name[,inf-section-name]...

This optional entry specifies the particular section(s) within the given INF file(s) that must be processed during the installation of this device. Typically, such a named section is a *DD-Install* (or *DDInstall.xxx*) section within one of the INF file(s) listed in an **Include** entry. However, it can be any section that is referenced within such a *DDInstall* or *DDInstall.xxx* section of the included INF.

For example, the INFs for device drivers that have the preceding **Include** entry specify this entry as **Needs= KS.Registration[, KSCAPTUR.Registration | KSCAPTUR. Registration.NT, MSPCLOCK.Installation]**

DelFiles=file-list-section[,file-list-section]...

This directive references one or more INF-writer-defined sections listing file(s) on the target to be deleted. In general, this directive is used only in INFs that upgrade a previous device/driver installation to remove obsolete files from the target machine.

RenFiles=file-list-section[,file-list-section]...

This directive references one or more INF-writer-defined sections listing file(s) to be renamed on the destination before device-relevant source files are copied to the target computer. Typically, this directive is used only in INFs that upgrade a previous installation to "save" previously installed files on the target machine from being overwritten.

DelReg=del-registry-section[,del-registry-section]...

This directive references one or more INF-writer-defined sections in which keys and/or value entries are specified to be removed from the registry during installation of the device(s).

Typically, this directive is used to handle upgrades when an INF must clean up old registry entries from a previous installation of this device. An **HKR** specification in such a delete-registry section designates the **..Class\***SetupClassGUID\device-instance-id* registry path to the user-accessible driver (a.k.a. "software" key).

BitReg=bit-registry-section[,bit-registry-section]...

This directive references one or more INF-writer-defined sections in which existing registry value entries of type REG_BINARY are modified. (See also **AddReg**.) An **HKR** specification in such a bit-registry section designates the **..Class\***SetupClassGUID\device-instance-id* registry path to the user-accessible driver (a.k.a. "software" key).

LogConfig=log-config-section[,log-config-section]...

This directive references one or more INF-writer-defined sections within an INF for a root-enumerated device or for a manually installed device. In these named sections, the INF for such a "detected" or manually installed device specifies one or more logical configurations of bus-relative hardware resources that the device must have to be operational. The INF for such a manually installed device that is not software-configurable also should have a *DDInstall***.FactDef** section.

The **LogConfig** directive is never used to install PnP peripheral devices, but, for more information about using a *DDInstall***.LogConfigOverride** section to override the hardware resource requirements reported by the underlying bus, see the reference for this directive later in this chapter.

This directive is irrelevant to all higher level (nondevice) drivers and components.

ProfileItems=profile-items-section[,profile-items-section]...

This directive references one or more INF-writer-defined sections that describe items to be added to, or removed from, the Start menu.

This directive is only supported on Windows 2000 platforms.

UpdateInis=update-ini-section[,update-ini-section]...

This rarely used directive references one or more INF-writer-defined sections, specifying a source INI file from which a particular section or line within such a section is to be read into a destination INI file of the same name during installation. Optionally, line-by-line modifications to an existing INI file on the destination from a given source INI file of the same name can be specified in the update-ini section.

UpdateIniFields=update-inifields-section[,update-inifields-section]...

This rarely used directive references one or more INF-writer-defined sections in which modifications within the lines of a device-specific INI file are specified.

Ini2Reg=ini-to-registry-section[,ini-to-registry-section]...

This rarely used directive references one or more INF-writer-defined sections in which sections or lines from a device-specific INI file, supplied on the source media, are to be moved into the registry.

Comments

The given *install-section-name* must be referenced in a device/models-specific entry under the per-manufacturer *Models* section of the INF file.

Except for devices that have no associated files to be transferred from the source media, a dual-OS INF file for a WDM driver must have at least two parallel *DDInstall* sections for a given device, as follows:

1. For Windows 9x platforms, provide an undecorated *DDInstall* section that specifies entries for device installations. There is no *DDInstall*.**Services** section in such a dual-OS INF file, because Windows 9x does not store the same information about device/driver services and dependencies in its registry as Windows 2000 does. Depending on the device, it can also have either or both of the optional *DDInstall*.**HW** and *DDInstall*.**Interfaces** sections to install the device/driver on Windows 9x platforms. (It might also have a *DDInstall*.**LogConfigOverride** section, as described later in the reference for the **LogConfig** directive.)
2. For Windows 2000 platforms, provide a corresponding *DDInstall*.**ntx86** section that specifies the entries for device/driver installations on x86-based Windows 2000 platforms. (Alternatively, the INF could have a corresponding *DDInstall*.**nt** section.) If the INF installs driver(s), such a dual-OS INF must have *DDInstall*.**ntx86.Services** section(s) to specify the device/driver registry information to be stored in the Windows 2000

registry's **...\CurrentControlSet\Services** tree. Depending on the device, it can also have one or more of the optional *DDInstall***.ntx86.HW**, *DDInstall***.ntx86.CoInstallers**, and/or *DDInstall***.ntx86.Interfaces** sections to install the same device/driver on x86-based Windows 2000 platforms. (It might also have a *DDInstall***.nt.LogConfigOverride** section, as described later in the reference for the **LogConfig** directive.)

For more information about how to use the system-defined **.nt** and **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File.*

Each directive in a *DDInstall* section can reference more than one section name, but each additional named section must be separated from the next with a comma (**,**). Each such section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files.*

Any **AddReg** directive specified in a *DDInstall* section is assumed to reference an add-registry section that cannot be used to store information about upper or lower filter drivers, about multifunction devices, or about driver-independent but device-specific parameters. If a device/driver INF must store this type of information in the registry, it must use an **AddReg** directive in its undecorated *DDInstall***.HW** section and decorated *DDInstall.xxx***.HW** sections, if any, to reference another INF-writer-defined add-registry section.

A *DDInstall* section can include a directive named **Reboot** or **Restart**. These directives are *only for compatibility with Windows 9x.* If one of these entries is present the OS is forced to reboot when the device is installed. These directives should never be used for PnP devices. In any case, it is best to let Setup determine whether the machine needs to be rebooted rather than specifying these directives.

Examples

This example shows the expansion of the *DDInstall* sections, Ser_Inst and Inp_Inst, referenced in the example for the immediately preceding per-manufacturer *Models* section.

```
[Ser_Inst]
CopyFiles=Ser_CopyFiles, mouclass_CopyFiles

[Ser_CopyFiles]
sermouse.sys

[mouclass_CopyFiles] ; section name referenced by > 1 CopyFiles
mouclass.sys

[Inp_Inst]
CopyFiles=Inp_CopyFiles, mouclass_CopyFiles

[Inp_CopyFiles]
inport.sys
```

This example shows the *DDInstall*.**NT** section in a dual-OS INF file for a system-supplied WDM driver of a particular audio device. With the exception of the **DriverVer** entry, its entries are identical to the undecorated *DDInstall* section for installations of the same device on Windows 9x platforms. However, a *DDInstall*.**NT** section is necessary in such a dual-OS INF file to set up the *DDInstall*.**NT.Services** section that is required to install the device/driver on Windows 2000 platforms.

```
[WDMPNPB003_Device.NT]
DriverVer=01/14/1999,5.0
Include=ks.inf, wdmaudio.inf
Needs=KS.Registration, WDMAUDIO.Registration.NT
LogConfig=SB16.LC1,SB16.LC2,SB16.LC3,SB16.LC4,SB16.LC5
; a few log-config-sections omitted here for brevity
CopyFiles=MSSB16.CopyList
AddReg=WDM_SB16.AddReg
```

The following shows the sections referenced by the preceding **Needs** entry in the system-supplied *ks.inf* and *wdmaudio.inf* files specified in the **Include** entry. When the Windows 2000 device installer and/or media class installer process this device's *DDInstall*.**NT** section, these next two sections are also processed.

```
[KS.Registration]
; following AddReg= is actually a single line in the ks.inf file
AddReg=ProxyRegistration,CategoryRegistration,\
  TopologyNodeRegistration,PlugInRegistration,PinNameRegistration,\
  DeviceRegistration
CopyFiles=KSProxy.Files,KSDriver.Files

[WDMAUDIO.Registration.NT]
AddReg=WDM.AddReg
CopyFiles=WDM.CopyFiles.Sys, WDM.CopyFiles.Drv
;
; INF-writer-defined add-registry and file-list sections
; referenced by preceding directives are omitted here for brevity
;
```

See Also

AddReg, **BitReg**, **CopyFiles**, **DelFiles**, **DelReg**, **DestinationDirs**, ***DDInstall*.CoInstallers**, ***DDInstall*.FactDef**, ***DDInstall*.HW**, ***DDInstall*.Interfaces**, ***DDInstall*.Services**, **Ini2Reg**, **LogConfig**, **Manufacturer**, ***Models***, **RenFiles**, **ProfileItems**, **SourceDisksFiles**, **Source-DisksNames**, **UpdateIniFields**, **UpdateInis**, **Version**

INF DDInstall.Services Section

[*install-section-name*.**Services**] |
[*install-section-name*.**nt.Services**] |
[*install-section-name*.**ntx86.Services**]

AddService=*ServiceName*,[*flags*],*service-install-section*[,
  event-log-install-section[,[*EventLogType*][,*EventName*]]]...
[**DelService=***ServiceName*[,[*flags*][,[*EventLogType*][,*EventName*]]]...
[**Include=***filename.inf*[,*filename2.inf*]...]
[**Needs=***inf-section-name*[,*inf-section-name*]...]

Each per-*Models DDInstall*.**Services** section contains one or more **AddService** directives referencing additional INF-writer-defined section(s) in a Windows 2000 INF file.

Windows 2000 INFs commonly use the *DDInstall*.**Services** section with at least one **Add-Service** directive to control how and when the services of a particular driver are loaded, any dependencies on other services or on underlying (legacy) drivers it might have, and so forth. Optionally, they set up event-logging services by the device driver(s) as well.

DDInstall.**Services** sections should have the same platform and OS decorations as their related *DDInstall* sections. For example, a *DDInstall*.**ntx86** section would have a corres-ponding *DDInstall*.**ntx86.Services** section.

This section is irrelevant to exclusively Windows 9x installations.

Directives and Entries

AddService=ServiceName,[flags],service-install-section[,event-log-install-section [,[EventLogType][,EventName]]]...

This directive references an INF-writer-defined *service-install-section* and, possibly, an *event-log-install-section* elsewhere in the INF file for the driver(s) of the device(s) covered by this *DDInstall* section.

DelService=ServiceName[,[flags][,[EventLogType][,EventName]]]...

This directive removes a previously installed service from the target machine. This directive is very rarely used, except possibly in an INF file that upgrades a previous installation of the same devices/models listed in the per-manufacturer per-*Models* section that defined the name of this *DDInstall* section.

Include=filename.inf[,filename2.inf]...

This optional entry specifies one or more additional named INF files containing sections needed to install this device. If this entry is specified, usually so is a **Needs** entry.

Needs=inf-section-name[,inf-section-name]...

This optional entry specifies the particular named section that must be processed during the installation of this device. Typically, such a named section is a *DDInstall*.**Services** or *DDInstall.xxx*.**Services** section within an INF file listed in an **Include** entry. However, it can be any section that is referenced within such a *DDInstall*.**Services** or *DDInstall. xxx*.**Services** section.

Comments

The given *DDInstall* section must be referenced in a device/models-specific entry under the per-manufacturer *Models* section of the INF file. The case-insensitive extensions to the *install-section-name* shown in the formal syntax statement can be inserted into such a *DD-Install*.**Services** section name in dual-OS and/or cross-platform INF files. For more information about how to use the system-defined **.nt** and **.ntx86** extensions in cross-platform Windows 2000 and/or dual-OS INF files, see also *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide.*

For more detailed information about INF-writer-defined *service-install-section*s and *event-log-install-section*s, see the reference for the **AddService** directive.

Examples

This example shows the *DDInstall*.**Services** section for the Ser_Inst section shown as an example in the immediately preceding reference for the *DDInstall* section.

```
[Ser_Inst.Services]
AddService=sermouse, 0x00000002, sermouse_Service_Inst,\
               sermouse_EventLog_Inst
;
; flags value in preceding entry indicates function driver of device
;
AddService = mouclass,, mouclass_Service_Inst, mouclass_EventLog_Inst

; entries in the following xxx_Inst sections omitted here for brevity,
; but fully specified as the example for the AddService directive
;
[sermouse_Service_Inst]
; ...

[sermouse_EventLog_Inst]
; ...

[mouclass_Service_Inst]
; ...

[mouclass_EventLog_Inst]
; ...
```

This example shows the *DDInstall*.**NT.Services** section and its INF-writer-defined *service-install-section*s in the dual-OS INF file for the system-supplied WDM audio device/driver shown as an example in the immediately preceding reference for the *DDInstall* section.

```
[WDMPNPB003_Device.NT.Services]
AddService = wdmaud,0x00000000,wdmaud_Service_Inst
AddService = swmidi,0x00000000,swmidi_Service_Inst
AddService = sb16,  0x00000002,sndblst_Service_Inst

[wdmaud_Service_Inst]
DisplayName   = %wdmaud.SvcDesc% ; friendly name (see Strings)
ServiceType   = 1                ; SERVICE_KERNEL_DRIVER
StartType     = 1                ; SERVICE_SYSTEM_START
ErrorControl  = 1                ; SERVICE_ERROR_NORMAL
ServiceBinary = %10%\system32\drivers\wdmaud.sys

[swmidi_Service_Inst]
DisplayName   = %swmidi.SvcDesc%
ServiceType   = 1
StartType     = 1
ErrorControl  = 1
ServiceBinary = %10%\system32\drivers\swmidi.sys

[sndblst_Service_Inst]
DisplayName   = %sndblst.SvcDesc%
ServiceType   = 1
StartType     = 1
ErrorControl  = 1
ServiceBinary = %10%\system32\drivers\mssb16.sys

[Strings] ; only immediately preceding %strkey% tokens shown here
%wdmaud.SvcDesc%="Microsoft WDM Virtual Wave Driver (WDM)"
%swmidi.SvcDesc%="Microsoft Software Synthesizer (WDM)"
%sndblst.SvcDesc%="WDM Sample Driver for SB16"
```

The reference for the *DDInstall*.**HW** section, next, has more examples of *DDInstall*.**Services** section(s) with some service-install section(s) referenced by the **AddService** directive, including one for a PnP filter driver.

See Also

AddService, ***DDInstall***, ***DDInstall.HW***, **DelService**, ***Models***

INF DDInstall.HW Section

[*install-section-name*.**HW**] |
[*install-section-name*.**nt.HW**] |
[*install-section-name*.**ntx86.HW**]

[**AddReg=***add-registry-section*[, *add-registry-section*]...] ...
[**Include=***filename.inf*[,*filename2.inf*]...]
[**Needs=***inf-section-name*[,*inf-section-name*]...]
[**DelReg=***del-registry-section*[, *del-registry-section*]...] ...
[**BitReg=***bit-registry-section*[,*bit-registry-section*] ...] ...

DDInstall.**HW** sections are typically used for installing multifunction devices, for installing PnP filter drivers, and for setting up any user-accessible device-specific but driver-independent information in the registry, whether with explicit **AddReg** directives or with **Include** and **Needs** entries.

Directives and Entries

AddReg=add-registry-section[, add-registry-section]...

References one or more INF-writer-defined *add-registry-section*s elsewhere in the INF file for the device(s) covered by this *DDInstall*.**HW** section. Such an add-registry section typically installs filters and/or stores per-device information in the registry. An **HKR** specification in such an add-registry section designates the **..Enum\***enumeratorID\device-instance-id* registry path to a user-accessible per-device (a.k.a. "hardware") subkey.

Include=filename.inf[,filename2.inf]...

Specifies one or more additional named INF files containing section(s) needed to install this device. If this entry is specified, usually so is a **Needs** entry.

Needs=inf-section-name[,inf-section-name]...

Specifies the named section(s) that must be processed during the installation of this device. Typically, such a named section is a *DDInstall*.**HW** (or *DDInstall.xxx*.**HW**) section within an INF file listed in an **Include** entry. However, it can be any section that is referenced within such a *DDInstall*.**HW** or *DDInstall.xxx*.**HW** section of the included INF.

DelReg=del-registry-section[, del-registry-section]...

References one or more INF-writer-defined *delete-registry-section*s elsewhere in the INF file for the driver(s) of the device(s) covered by this *DDInstall* section. Such a delete-registry section removes stale registry information for a previously installed device/driver from the target machine. An **HKR** specification in such a delete-registry section designates the same subkey as for **AddReg**.

This directive is rarely used, except in an INF file that upgrades a previous installation of the same devices/models listed in the per-manufacturer per-*Models* section that defined the name of this *DDInstall* section.

BitReg=bit-registry-section[,bit-registry-section]...

Is valid in this section, but almost never used. An **HKR** specification in a referenced bit-registry section designates the same subkey as for **AddReg**.

Comments

The case-insensitive extensions to the *install-section-name* shown in the formal syntax statement can be inserted into such a *DDInstall*.**HW** section name in cross-platform Windows 2000 and/or dual-OS INF files. For more information about how to use the system-defined **.nt** and **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File.*

Any *DDInstall*.**HW** section either must have an **AddReg** directive or must **Include** another INF file and reference a section in the corresponding **Needs** entry that sets up the necessary registry information.

Each directive in a *DDInstall*.**HW** section can reference more than one INF-writer-defined section, but each additional named section must be separated from the next with a comma (**,**). Each such section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files.*

For more information about installing multifunction devices, see also the *Plug and Play, Power Management, and Setup Design Guide.*

Example

This example shows how the Windows 2000 CD-ROM device class installer INF uses *DDInstall*.**HW** sections to support both CD audio and changer functionality by creating the appropriate registry section(s) and setting these up as PnP upper filter drivers with **AddService** directives.

```
;;
;; Installation section for cdaudio. Sets cdrom as the service
;; and adds cdaudio as a PnP upper filter driver.
;;
[cdaudio_install]
CopyFiles=cdaudio_copyfiles,cdrom_copyfiles

[cdaudio_install.HW]
AddReg=nosync_addreg,cdaudio_addreg
 ; cdaudio_addreg required to register this as a PnP filter driver
```

```
[cdaudio_install.Services]
AddService=cdrom,0x00000002,cdrom_ServiceInstallSection
AddService=cdaudio,,cdaudio_ServiceInstallSection

[changer_install]
CopyFiles=changer_copyfiles,cdrom_copyfiles

[changer_install.HW]
AddReg=changer_addreg

; ... changer_install.Services section similar to cdaudio's

; ... some similar cdrom_install(.HW)/addreg sections omitted

[cdaudio_addreg] ; changer_addreg section has similar entry
HKR,,"UpperFilters",0x00010000,"cdaudio" ; REG_MULTI_SZ value

;
; Use next section to disable synchronous transfers to this device.
; Sync transfers will always be turned off by default in this INF
; for any cdrom-type device.
;
[nosync_addreg]
HKR,,"DefaultRequestFlags",0x00010001,8

[autorun_addreg]
HKLM,"System\CurrentControlSet\Services\cdrom","AutoRun",0x00010003,1
;;
;; service-install sections for cdrom, cdaudio, and changer
;;
[cdrom_ServiceInstallSection]
DisplayName    = %cdrom_ServiceDesc%
ServiceType    = 1
StartType      = 1
ErrorControl   = 1
ServiceBinary  = %12%\cdrom.sys
LoadOrderGroup = SCSI CDROM Class
AddReg         = autorun_addreg

[cdaudio_ServiceInstallSection]
DisplayName    = %cdaudio_ServiceDesc%
ServiceType    = 1
StartType      = 1
ErrorControl   = 1
ServiceBinary  = %12%\cdaudio.sys

; ... changer_ServiceInstallSection similar to cdaudio's
```

See Also

AddReg, **BitReg**, ***DDInstall***, ***DDInstall*.Services**, **DelReg**

INF DDInstall.CoInstallers Section

[*install-section-name*.**CoInstallers**] |
[*install-section-name*.**nt.CoInstallers**] |
[*install-section-name*.**ntx86.CoInstallers**]

AddReg=*add-registry-section*[, *add-registry-section*]...
CopyFiles=@*filename* | *file-list-section*[,*file-list-section*]...
[**Include=***filename.inf*[,*filename2.inf*]...]
[**Needs=***inf-section-name*[,*inf-section-name*]...]
[**DelFiles=***file-list-section*[,*file-list-section*]...]
[**RenFiles=***file-list-section*[,*file-list-section*]...]
[**DelReg=***del-registry-section*[, *del-registry-section*]...]
[**BitReg=***bit-registry-section*[,*bit-registry-section*]...]
[**UpdateInis=***update-ini-section*[,*update-ini-section*]...]
[**UpdateIniFields=***update-inifields-section*[,*update-inifields-section*]...]
[**Ini2Reg=***ini-to-registry-section*[,*ini-to-registry-section*]...]
...

This optional section registers one or more device-specific coinstallers or (rarely) device-class-specific coinstallers supplied on the distribution media to supplement the operations of existing device class installer(s).

Directives and Entries

AddReg=add-registry-section[, add-registry-section]...

References one or more INF-writer-defined *add-registry-section*s that store registry information about the supplied coinstaller(s).

An **HKR** specified in such an add-registry section designates the **..Class\***SetupClassGUID\\ device-instance-id* registry path to the user-accessible driver (a.k.a. "software") key. Thus, for a device-specific coinstaller, it writes (or modifies) a **CoInstallers32** value entry in this user-accessible per-device/driver "software" subkey.

For a class-specific coinstaller, it registers the new coinstaller(s) by modifying the contents of the appropriate **..CoDeviceInstallers\***SetupClassGUID* subkey(s). The path to the appropriate registry *SetupClassGUID* subkey(s) must be explicitly specified in the referenced add-registry section(s).

CopyFiles=@filename | file-list-section[,file-list-section]...

Transfers the source coinstaller file(s) to the destination on the target machine, usually by referencing one or more INF-writer-defined *file-list-sections* elsewhere in the INF file. Such a file-list section specifies the coinstaller file(s) to be copied from the source media to the destination directory on the target.

However, system INFs that install coinstallers never use this directive in a *DDInstall.* **CoInstallers** section.

Include=filename.inf[,filename2.inf]...

Specifies one or more additional named INF files containing sections needed to install the coinstaller(s) for this device or device setup class. If this entry is specified, usually so is a **Needs** entry.

Needs=inf-section-name[,inf-section-name]...

Specifies the particular section(s) that must be processed during the installation of this device. Typically, such a named section is a *DDInstall*.**CoInstallers** (or *DDInstall.xxx*.-**CoInstallers**) section within a INF file listed in an **Include** entry. However, it can be any section that is referenced within such a *DDInstall*.**CoInstallers** or *DDInstall.xxx*.-**CoInstallers** section of the included INF.

DelFiles=file-list-section[,file-list-section]...

References a file-list section specifying file(s) to be removed from the target. This directive is rarely used, but it might be used in an INF that upgrades a previous installation with new coinstaller file(s).

RenFiles=file-list-section[,file-list-section]...

References a file-list section specifying file(s) on the destination to be renamed before co-installer source files are copied to the target.This directive also is rarely used, but it might be used in an INF that upgrades a previous installation with new coinstaller file(s).

DelReg=del-registry-section[, del-registry-section]...

References one or more INF-writer-define *delete-registry-sections*. Such a section specifies stale registry information about the coinstaller(s) for a previous installation of the same device(s) that should be removed from the registry. An **HKR** specified in such a delete-registry section designates the same registry subkey as already described for **AddReg**. This directive is very rarely used in a *DDInstall*.**CoInstallers** section.

BitReg=bit-registry-section[,bit-registry-section]...

Is valid in this section but almost never used. An **HKR** specified in such a bit-registry section designates the same registry subkey as already described for **AddReg**.

UpdateInis=update-ini-section[,update-ini-section]...

Is valid in this section but almost never used.

UpdateIniFields=update-inifields-section[,update-inifields-section]...

Is valid in this section but almost never used.

Ini2Reg=ini-to-registry-section[,ini-to-registry-section]...

Is valid in this section but almost never used.

Comments

The given *DDInstall* section must be referenced in a device/models-specific entry under the per-manufacturer *Models* section of the INF file.

If an INF includes a *DDInstall*.**Coinstallers** section, there must be one for each platform-decorated and undecorated *DDInstall* section. For example, if an INF contains an [*install-section-name*.**ntx86**] section and an [*install-section-name*] section and it registers device-specific coinstaller(s), then the INF must include both an [*install-section-name*.**ntx86. Coinstallers**] section and an [*install-section-name*.**Coinstallers**] section. For more information about how to use the system-defined **.nt** and **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File*. Coinstallers are not supported on Windows 9x platforms.

Each directive in a *DDInstall*.**CoInstallers** section can reference more than one INF-writer-defined section name, but each additional named section must be separated from the next with a comma (,). Each directive-created section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files.*

A coinstaller is a Win32 DLL that typically writes additional configuration information to the registry or performs other installation tasks that require dynamically generated, machine-specific information that is not available when an INF is created. A device-specific co-installer supplements the installation operations either of the OS's device installer or of the appropriate class installer when that device is installed. A device-class-specific coinstaller supplements the installation operations either of the OS's device installer or of the appropriate class installer for every device of that class when they are installed.

For more information about writing and using coinstallers, see also the *Plug and Play, Power Management, and Setup Design Guide.*

Installing Coinstaller Images

All coinstaller files must be copied into the *%windir%\system32* directory on Windows 2000 machines, or into *%windir%\system* on Windows 9x machines. Like any INF **CopyFiles** operation, the destination is controlled explicitly for a named *file-list-section* in the

DestinationDirs section of the INF file by the *dirid* value **11** or by supplying this *dirid* value for the **DefaultDestDir** entry.

Registering Device-Specific Coinstallers

To add the REG_MULTI_SZ-type value entry for one or more device-specific coinstallers to the registry, an *add-registry-section* referenced by the **AddReg** directive has the following general form:

[*DDInstall*.**CoInstallers**_*DeviceAddReg*]

HKR,,CoInstallers32,0x00010000,"*DevSpecificCoInstall*.**dll**
  [,*DevSpecificEntryPoint*]"[,"*DevSpecific2CoInstall*.**dll**
    [,*DevSpecific2EntryPoint*]"...]

The entry in such an add-registry section appears as a single line within the INF file, and each supplied device-specific coinstaller DLL must have a unique name. After it has been registered, the system's device installer calls such a device-specific coinstaller at each subsequent step of the installation process for that device.

When the optional *DevSpecificEntryPoint* is omitted from an add-registry section entry, the default CoDeviceInstall routine is assumed to be the entry point of any coinstaller DLL.

Registering Device-Class Coinstallers

To add a value entry (and setup-class subkey, if it does not exist already) for one or more device-class coinstallers to the registry, an *add-registry-section* referenced by the **AddReg** directive has the following general form:

[*DDInstall*.**CoInstallers**_*ClassAddReg*]

HKLM,System\CurrentControlSet\Control
  \CoDeviceInstallers,{*SetupClassGUID*}**,**
    0x00010008,"*DevClssCoInst*.**dll**[,*DevClssEntryPoint*]"
...

Each entry in such an add-registry section appears as a single line within the INF file, and each supplied class coinstaller DLL must have a unique name. If the supplied coinstaller(s) should be used for more than one class of devices, this add-registry section can have more than one entry, each with the appropriate *SetupClassGUID* value. (The reference for the **Version** section earlier in this chapter contains a summary of the system-defined device classes and their respective setup *ClassGUID* values.)

Such a supplemental device-class coinstaller must not replace any already registered coinstaller(s) for an existing class installer, so it must have a unique name and the REG_MULTI_SZ-type value supplied must be appended (as indicated by the **8** in the *flags* value **0x0010008**) to the class-specific coinstaller entries, if any, already present in the

{*SetupClassGUID*} subkey. However, the OS setup functions will never append a duplicate *DevClssCoInstall*.**dll** to a value entry if a coinstaller of the same name is already registered.

The INF for a supplemental device-class coinstaller can be activated by a right-click install or through a custom setup application's call to **SetupInstallFromInfSection**.

Example

This example shows the *DDInstall*.**CoInstallers** section for IrDA serial network cards. The system-supplied INF for these IrDA (serial) NICs supplies a coinstaller to the system IrDA class installer.

```
; DDInstall section
[PNP.NT]
AddReg=ISIR.reg, Generic.reg, Serial.reg
PromptForPort=0     ; This is handled by IRCLASS.DLL
LowerFilters=SERIAL ; This is handled by IRCLASS.DLL
BusType=14
Characteristics=0x4 ; NCF_PHYSICAL

; ... PNP.NT.Services section omitted here
[PNP.NT.CoInstallers]
AddReg = ISIR.CoInstallers.reg
; ...
[IRSIR.reg]
HKR, Ndi, HelpText, 0, %IRSIR.Help%
HKR, Ndi, Service, 0, "IRSIR"
HKR, Ndi\Interfaces, DefUpper, 0, "ndisirda"
HKR, Ndi\Interfaces, DefLower, 0, "nolower"
HKR, Ndi\Interfaces, UpperRange, 0, "ndisirda"
HKR, Ndi\Interfaces, LowerRange, 0, "nolower"

[Generic.reg]
HKR,,InfraredTransceiverType,0,"0"

[Serial.reg]
HKR,,SerialBased,0, "0"

[ISIR.CoInstallers.reg]
HKR,,CoInstallers32,0x00010000,"IRCLASS.dll,IrSIRClassCoInstaller"

; ... Services and Event Log registry sections omitted here
[Strings]
; ...
IRSIR.Help = "An IrDA serial infrared device is a built-in COM port or
external transceiver which transmits infrared pulses. This NDIS
miniport driver installs as a network adapter and binds to the FastIR
protocol."
```

The preceding PNP.**NT.CoInstallers** section only referenced a coinstaller-specific add-registry section. It has no **CopyFiles** directive because this system-supplied INF installs a set of IrDA network devices and, like all system INFs, uses the **LayoutFile** entry in its **Version** section to transfer the coinstaller file to the destination. However, any *DDInstall.* **CoInstallers** section in an INF supplied by an IHV or OEM also would have a **CopyFiles** directive, along with **SourceDisksNames** and **SourceDisksFiles** sections.

See Also

AddReg, BitReg, CopyFiles, *DDInstall*, **DelFiles, DelReg, DestinationDirs, Ini2Reg, RenFiles, SourceDisksFiles, SourceDisksNames, UpdateIniFields, UpdateInis, Version**

INF DDInstall.Interfaces Section

[*install-section-name*.**Interfaces**] |
[*install-section-name*.**nt.Interfaces**] |
[*install-section-name*.**ntx86.Interfaces**]

AddInterface={*interfaceGUID*} [, [*reference string*] [,[*add-interface-section*] [,*flags*]]] ...
[**Include=***filename.inf*[,*filename2.inf*]...]
[**Needs=***inf-section-name*[,*inf-section-name*]...]

Each per-Models *DDInstall*.**Interfaces** section can have one or more **AddInterface** directives, depending on how many predefined device interfaces a particular device/driver supports and/or how many new (and compatible with existing) device interfaces the driver(s) of that device export for use by still higher level components.

To support existing device interfaces, such as any of the system's predefined kernel-streaming interfaces, specify the appropriate *interfaceGUID* value(s) in this section. To install a component, such as a class driver, that exports a whole new class of device interfaces, an INF also must have an **InterfaceInstall32** section.

Directives and Entries

AddInterface={interfaceGUID} [, [reference string] [,[add-interface-section] [,flags]]] ...

This directive installs support for a device interface, designated by the given *interfaceGUID* value, that the device/driver exports to higher level components. Usually, it also references an INF-writer-defined *add-interface-section* elsewhere in the INF file.The {*interfaceGUID*} and/or (rarely specified) *reference-string* can be expressed as *%strkey%* tokens that are defined in a **Strings** section of the INF file. The INF of a PnP device function or filter driver usually omits the optional *reference string* entry unless that driver uses reference strings to discriminate between two interfaces of the same class for a single device. The *flags*, if specified, must be zero.

An *add-interface-section* typically references an add-registry section that contains value entries to be stored in the registry about the interface(s) supported by the device/driver. An **HKR** specified in such an add-registry section designates the user-accessible device-interface subkey of the **..DeviceClasses\{***InterfaceClassGuid***}** registry branch. In the Windows 2000 registry, **DeviceClasses** is a subkey of the **..CurrentControlSet\Control** key.

Include=filename.inf[,filename2.inf]...

This optional entry specifies one or more additional named INF files containing sections needed to register the interfaces supported by this device/driver. If this entry is specified, usually so is a **Needs** entry.

Needs=inf-section-name[,inf-section-name]...

This optional entry specifies the particular section(s) that must be processed during the installation of this device. Typically, such a named section is a *DDInstall***.Interfaces** (or *DDInstall.xxx***.Interfaces**) section within a INF file listed in an **Include** entry. However, it can be any section that is referenced within such a *DDInstall***.Interfaces** or *DDInstall.xxx***. Interfaces** section of the included INF.

Comments

The given *DDInstall* section must be referenced by a device/models-specific entry under the per-manufacturer *Models* section of the INF file. The case-insensitive extensions to the *install-section-name* shown in the formal syntax statement can be inserted into such a *DDInstall***.Interfaces** section name in cross-platform and/or dual-OS INF files. For more information about how to use the system-defined **.nt** or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File.*

If a given {*interfaceGUID*} is not installed already, the OS's setup code installs that device interface class in the system. Usually, an INF that installs one or more new device interfaces for every subsequently installed device also has an **[InterfaceInstall32]** section containing each specified {*interfaceGUID*} as an entry to set up registry information, copy any necessary files, and so forth for such new device-interface classes.

For more information about how to create a GUID, see the Platform SDK documentation. For the system-defined interface class GUIDs, see the appropriate system-supplied header, such as *ks.h* for the kernel-streaming interface class GUIDS.

When a driver is loaded, it must call **IoSetDeviceInterfaceState** once with each {*interface-GUID*} value specified in the INF's *DDInstall***.Interfaces** section that the driver supports on the underlying device to enable the interface for runtime use by higher level components. As an alternative to registering its support for a device interface in its INF, a device driver can call **IoRegisterDeviceInterface** before making its initial call to **IoSetDeviceInterfaceState**. Usually, a PnP function or filter driver makes this call from its AddDevice routine.

Each **AddInterface** directive in a *DDInstall*.**Interfaces** section can reference an INF-writer-defined *add-interface-section*. Each such section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*.

For more detailed information about INF-writer-defined *add-interface-section*s, see the reference for the **AddInterface** directive.

Example

This example shows the *DDInstall*.**NT.Interfaces** section in the dual-OS INF file for the system-supplied WDM audio device/driver shown as examples in the preceding references for the *DDInstall* section and *DDInstall*.**Services** section.

```
;
; following AddInterface= are all single lines (without
; backslash line continuators) in the system-supplied INF file
;
[WDMPNPB003_Device.NT.Interfaces]
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_Wave%,\
  WDM_SB16.Interface.Wave
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_Topology%,\
  WDM_SB16.Interface.Topology
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_UART%,\
  WDM_SB16.Interface.UART
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_FMSynth%,\
  WDM_SB16.Interface.FMSynth
; ...

[Strings] ; only immediately preceding %strkey% tokens shown here
%KSCATEGORY_AUDIO% = "{6994ad04-93ef-11d0-a3cc-00a0c9223196}"
KSNAME_Wave = "Wave"
KSNAME_UART = "UART"
KSNAME_FMSynth = "FMSynth"
KSNAME_Topology = "Topology"
; ...
```

See Also

AddInterface, ***DDInstall***, **InterfaceInstall32**, **IoRegisterDeviceInterface**, **IoSetDeviceInterfaceState**

INF InterfaceInstall32 Section

[InterfaceInstall32]

{*InterfaceClassGUID*}=*install-interface-section*[,*flags*]... |
install-interface-section.**nt**[,*flags*]... |
install-interface-section.**ntx86**[,*flags*]...

...

This section sets up one or more new device interface classes for a device/driver that exports such an interface to still higher level components. The INF of such a device/driver also has a *DDInstall*.**Interfaces** section to register its own support for each such new device interface class or else the driver must call **IoRegisterDeviceInterface** with each *InterfaceClassGUID* value when it is loaded.

Note that any subsequently installed devices/drivers also can register their support for such a new device interface in the *DDInstall*.**Interfaces** sections of their respective INF files or by calling **IoRegisterDeviceInterface**. In effect, this section bootstraps a new device interface class for all devices/drivers that register support for that interface.

Entry Values

InterfaceClassGUID

Specifies a GUID value identifying the newly exported device interface.

Each given GUID value in this section also must be referenced by an **AddInterface** directive in the *DDInstall*.**Interfaces** (or appropriate *DDInstall.xxx*.**Interfaces**) section of the INF file, or else the newly installed device's driver must call **IoRegisterDeviceInterface** with this GUID.

For more information about how to create a GUID, see the Platform SDK documentation. For the system-defined interface class GUIDS, see the appropriate headers, such as *ks.h* for the kernel-streaming interfaces.

install-interface-section

References an INF-writer-defined section, possibly with any of the system-defined extensions, elsewhere in this INF.

flags

If specified, must be zero.

Comments

When a given *InterfaceClassGUID* is not already installed in the system, that interface is installed as the corresponding *DDInstall*.**Interfaces** (or *DDInstall.xxx*.**Interfaces**) section is processed by the system setup functions during device installation or when that device's driver makes the initial call to **IoRegisterDeviceInterface**.

Each INF-writer-created *interface-install-section* name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*. Extensions, such as **.NT** or **.ntx86**, to a given *interface-install-section* allow OS-specific and/or platform-specific interfaces to be exported from the same INF file. For more information about these system-defined extensions, see *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide*.

Any given *interface-install-section* has the following general form:

[*interface-install-section*]

AddReg=*add-registry-section*[, *add-registry-section*] ...
[**Copyfiles=**@*filename* | *file-list-section*[,*file-list-section*] ...]
[**DelReg=***del-registry-section*[, *del-registry-section*] ...]
[**BitReg=***bit-registry-section*[,*bit-registry-section*]...]
[**Delfiles=***file-list section*[,*file-list-section*] ...]
[**Renfiles=***file-list-section*[,*file-list-section*] ...]
[**UpdateInis=***update-ini-section*[,*update-ini-section*]...]
[**UpdateIniFields=***update-inifields-section*[,*update-inifields-section*]...]
[**Ini2Reg=***ini-to-registry-section*[,*ini-to-registry-section*]...]
...

As shown here, an interface-install section must have at least one **AddReg** directive that references one or more add-registry sections to set up device-interface-specific information in the registry during installation of this interface. An **HKR** specified in such an add-registry section designates the **..DeviceClasses\\{***InterfaceClassGUID***}** key.

The registry information about this interface should include at least a friendly name for the new device interface and whatever information the higher level components that open and use this interface will need.

In addition, such an *interface-install-section* might use any of the optional directives shown here to specify interface-specific installation operations.

See Also

AddReg, **BitReg**, **ClassInstall32**, **CopyFiles**, ***DDInstall***, ***DDInstall*.Interfaces**, **DelFiles**, **DelReg**, **Ini2Reg**, **IoRegisterDeviceInterface**, **RenFiles**, **UpdateIniFields**, **UpdateInis**

INF DDInstall.FactDef Section

[*install-section-name*.**FactDef]** |
[*install-section-name*.**nt.FactDef]** |
[*install-section-name*.**ntx86.FactDef]**

ConfigPriority=*Priority_Value*
[**DMAConfig=**[*DMAattrs*:]*DMANum*]
[**IOConfig=***io-range*]
[**MemConfig=***mem-range*]
[**IRQConfig=**[*IRQattrs*:]*IRQNum*
...

This section should be used in an INF for any manually installed nonPnP device that an end user might install with the Add New Hardware wizard. This section specifies the factory-default hardware configuration settings, such as the bus-relative I/O ports, IRQ (if any), and so forth, for such a card.

Section Entries and Values

ConfigPriority=Priority_Value

Specifies the priority value for this factory-default logical configuration, as HARD-RECONFIG, indicating a jumper change is required to reset this logical configuration.

DMAConfig=[DMAattrs:]DMANum

Specifies the bus-relative DMA channel as a decimal number. *DMAattrs* is optional if the device is connected on a bus that has only 8-bit DMA channels and the device uses standard system DMA. Otherwise, it can be one of the letters **D** for 32-bit DMA, **W** for 16-bit DMA, and **N** for 8-bit DMA, with **M** if the device uses busmaster DMA and with one of the following (mutually exclusive) letters indicating the type of DMA channel used: **A**, **B**, or **F**. If none of **A**, **B**, or **F** is specified, a standard DMA channel is assumed.

IOConfig=io-range

Specifies the I/O port range for the device in the following form:

start-end[([decode-mask][:alias-offset][:attr])]
where:

start specifies the (bus-relative) starting address of the I/O port range as a 64-bit hexadecimal value.

end specifies the ending address of the I/O port range, also as an 64-bit hexadecimal value.

decode-mask defines the alias type and can be any of the following:

Mask Value	Meaning	IOR_Alias =
3ff	10-bit decode	0x04
fff	12-bit decode	0x10
ffff	16-bit decode	0x00
0	– positive decode	0xFF

alias-offset is ignored.

attr specifies the letter **M** if the given range is in system memory. If omitted, the given range is in I/O port space.

MemConfig=mem-range

Specifies the memory range for the device in the following form:

start-end[(*attr*)]
where:

start specifies the starting (bus-relative) address of the device memory range as a 64-bit hexadecimal value.

end specifies the ending address of the memory range, also as a 64-bit hexadecimal value.

attr specifies the attributes of the memory range as one or more of the following letters: **R** (read-only), **W** (write-only), **RW** (read/write), **C** (combined write allowed), **H** (cacheable), **F** (prefetchable), and **D** (card decode addressing is 32-bit, instead of 24-bit) If both **R** and **W** are specified or if neither is specified, read/write is assumed.

IRQConfig=[IRQattrs:]IRQNum

Specifies the bus-relative IRQ that the device uses as a decimal number. *IRQattrs* is omitted if the device uses a bus-relative, edge-triggered IRQ. Otherwise, specify **L** to indicate a level-triggered IRQ and **LS** if the device can share the IRQ line listed in this entry.

Comments

The given *DDInstall* section must be referenced in a device-specific entry under the per-manufacturer *Models* section of the INF file. The case-insensitive extensions to the *install-section-name* shown in the formal syntax statement can be inserted into such a *DDInstall*.**FactDef** section name in cross-OS and/or cross-platform INF files. For more information about these system-defined extensions, see *Creating an INF File.*

This section must contain complete factory-default information for installing one device. The INF should specify this set of entries in the order best suited to how the driver initializes its device. If necessary, it can have more than one of any particular kind of entry. For example, the INF for a device that used two DMA channels would have two **DMAConfig=**

lines in its *DDInstall*.**FactDef** section. From this section, the Add New Hardware wizard builds binary logical configuration records and stores them in the registry.

The INF files of manually installed devices for which the factory-default logical configuration setting(s) can be changed also should use the **LogConfig** directive in their *DDInstall* sections. In general, such an INF should specify the entries in each of its log-config sections and in its *DDInstall*.**FactDef** section in the same order.

Examples

This **IOConfig=** entry specifies an I/O port region, eight bytes in size, which can start at 2F8.

```
IOConfig=2F8-2FF
```

This **MemConfig=** entry specifies a memory region of 32K bytes that can start at D0000.

```
MemConfig=D0000-D7FFF
```

See Also

DDInstall, **LogConfig**

INF Strings Section

[Strings] | **[Strings.***LanguageID***]** ...

strkey1= ["]*some string*["]
strkey2 = "   *string-with-leading-or-trailing-whitespace*   " |
        "*very-long-multiline-string* " |
        "*string-with-semicolon*" | "*string-ending-in-backslash*" |
        ""*double-quoted-string-value*""
...

An INF file must have at least one **Strings** section to define every *%strkey%* token specified elsewhere in that INF.

Entry Values

strkey1, strkey2, ...

Each *strkey* in an INF file must specify a unique name consisting of letters, digits, and/or other explicitly visible characters. A % character within such a *strkey* token can be expressed as *%%*.

some string | "some string"

Specifies a string, optionally delimited with double-quote characters ("), that contains letters, digits, punctuation, and possibly even certain implicitly visible characters, in particular, internal space and/or tab characters. However, an unquoted string cannot contain an internal double-quote ("), semicolon (;), linefeed, return, or any invisible control characters, and it cannot have a backslash (\) as its final character.

" string-with-leading-or-trailing-whitespace " | "very-long-multiline-string" | "string-with-semicolon" | "string-ending-in-backslash" | ""double-quoted-string-value""

Any given string containing leading or trailing whitespace(s) that is so long it linewraps, or a string that contains a semicolon or a final backslash character must be be enclosed in a pair of double-quote characters ("). The system INF parser discards the outermost enclosing pair of double-quote characters delimiting such a string, along with any leading or trailing whitespace characters outside the double-quote string delimiters. To summarize, the value specified for a *%strkey%* token *must* be enclosed in double quotes, if it meets any of the following criteria:

- If a given string has leading and/or trailing whitespace(s) that must be retained as part of its value, that string must be enclosed in double-quote characters to prevent its leading and/or trailing whitespaces from being discarded by the INF parser.
- If a long string might contain any internal linefeed or return characters due to line-wrapping in the text editor, it also should be enclosed in double-quotes to prevent truncation of the string at the initial internal linefeed or return character.
- If such a string contains a semicolon, it must be enclosed in double-quotes to prevent the string from being truncated at the semicolon. (As already mentioned in *General Syntax Rules for INF Files*, the semicolon character begins each comment in INF files.)
- If such a string ends in a backslash, it must be enclosed in double-quotes to prevent the string from being concatenated with the next entry. (As already mentioned in *General Syntax Rules for INF Files*, the backslash character (\) is used as the line continuator in INF files.)
- Like an unquoted string specification, such a "*quoted string*" cannot contain internal double-quote characters. However, it can be specified as an explicitly double-quoted string value by using one or more additional pairs of double-quote characters (for example, """*some string*""").

The Windows 2000 INF parser not only discards the outermost pair of enclosing double-quotes for any "*quoted string*" in this section, but also condenses each subsequent sequential pair of double-quotes into a single double-quote character. That is, """"*some string*"""" also becomes "*some string*" when it has been parsed.

Comments

Because the system INF parser strips the outermost pair of enclosing double quotes from any "*quoted string*" defining a *%strkey%* token, many of the system INFs define all their *%strkey%* tokens as "*quoted string*"s to avoid the inadvertant loss of leading and trailing whitespaces during INF parsing, to ensure that particularly long string values that wrap across lines cannot be truncated, and to ensure that strings with ending backslashes cannot be concatenated to the next line in the INF file.

To create a single international INF file, the INF can have a set of locale-specific **Strings.** *LanguageID* sections, as shown in the formal syntax statement. The *LanguageID* extension is a hexadecimal value whose lower 10 bits contain the primary language ID and the next 6 bits contain the sublanguage ID. The language and sublanguage IDs match the system-defined values of the Win32 LANG_*XXX* and SUBLANG_*XXX* constants. (See the Platform SDK's *winnt.h* file for the definition of these constants.) For example, a *LanguageID* value of 0407 represents a primary language ID of LANG_GERMAN (07) with a sublanguage ID of SUBLANG_GERMAN (01).

Depending on the current locale of a particular machine, the system setup functions process each such **Strings.***LanguageID* section in a given INF file as follows:

1. First, look for the *.LanguageID* values in the INF that match the current locale assigned to the machine. If such an exact match is found, use that Strings.LanguageID INF section to translate %strkey% tokens for this device/driver installation.
2. Otherwise, look next for a match to the LangID value with the value of SUBLANG_ NEUTRAL as the SubLangID. If such a match is found, use that INF section to translate %strkey% tokens for this device/driver installation.
3. Otherwise, look next for a match to the LangID value and any valid SubLangID for the same LangID family. If such a partial match is found, use that Strings.LanguageID INF section to translate %strkey% tokens for this device/driver installation.
4. Otherwise, use the undecorated Strings section(s) of the INF to translate %strkey% tokens for this installation.

By convention, the **Strings** section is the last within all system INF files for convenience in creating a set of INFs for the international market. Using *%strkey%* tokens for all user-visible string values within an INF simplifies the translation(s) of such strings by placing all that can be displayed to the end user in a single per-locale **Strings** section of an INF file.

Although the **Strings** section is usually the last section in every INF file, any given *%strkey%* token defined in the **Strings** section can be used repeatedly elsewhere in the INF, in particular, wherever the "translated" value of that token is required. The system setup functions expand each *%strkey%* token to the specified string and then use that expanded value for further INF processing.

The use of *%strkey%* tokens within INF files is not restricted to user-visible string values. These tokens can be used in any manner convenient to the INF writer, as long as each such token is defined within a **Strings** section of the INF file. For example, when writing an INF file that requires the specification of several GUIDs, it might be convenient to use a per-GUID number of *%strkey%* tokens with meaningful names as substitutes for each such GUID value. Specifying such a set of %strkey% = "{*GUID*}" values in the INF file's **Strings** section would require explicit GUID values to be typed only once each, and it would provide more human-readable internal INF documentation than using explicit GUID values throughout the INF file.

All *%strkey%* tokens must be defined within the INF file in which they are referenced. This implies the following for any INF file that has **Include=** and **Needs=** entries in one or more of its *DDInstall*, *DDInstall*.**Services**, *DDInstall*.**HW**, *DDInstall*.**CoInstallers**, and/or *DDInstall*.**Interfaces** sections:

- An included INF is assumed to have its own respective **Strings** section to define all *%strkey%* tokens referenced in that INF.

Example

The following example shows a fragment of a **Strings** section from a system-supplied locale-specific *dvd.inf* for installations in English-speaking countries.

```
[Strings]
Msft="Microsoft"
MfgToshiba="Toshiba"
Tosh404.DeviceDesc="Toshiba DVD decoder card"
; ...
```

See Also

DDInstall, ***DDInstall*.CoInstallers**, ***DDInstall*.HW**, ***DDInstall*.Interfaces**, ***DDInstall*.Services**, **Manufacturer**, **InterfaceInstall32**, ***Models***, **SourceDisksNames**, **Version**

INF AddReg Directive

[*DDInstall*] | [*DDInstall*.**HW**] | [*DDInstall*.**CoInstallers**] | **[ClassInstall32]** | **[ClassInstall32.ntx86]**

AddReg=*add-registry-section*[**,** *add-registry-section*] ...

An **AddReg** directive references one or more INF-writer-defined sections used to modify or create registry information. Each such add-registry section can have entries to do the following:

- Add new keys, possibly with initial value entries, to the registry

- Add new value entries to existing registry keys
- Modify existing value entries of particular keys in the registry

An **AddReg** directive can be specified under any of the sections shown in the formal syntax statement. This directive also can be specified under any of the following INF-writer-defined sections:

- A *service-install-section* or *event-log-install* section referenced by the **AddService** directive in a *DDInstall*.**Services** section
- An *add-interface-section* referenced by the **AddInterface** directive in a *DDInstall*.**Interfaces** section
- An *install-interface-section* referenced in an **InterfaceInstall32** section

Comments

Any given *add-registery-section* name must be unique to the INF file, but it can be referenced by **AddReg** directives in other sections of the same INF. Each INF-writer-defined section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files.* For more information about how to use the system-defined **.nt** or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File.*

Each named section referenced by an **AddReg** directive has the following form:

[*add-registry-section*]

reg-root, [*subkey*], [*value-entry-name*], [*flags*], [*value*]
...

An *add-registy-section* section can have any INF-writer-determined number of entries, each on a separate line.

AddReg-Referenced Section Entries

reg-root

Identifies the root of the registry tree for other values supplied in this entry. The value can be one of the following:

HKCR

Abbreviation for **HKEY_CLASSES_ROOT**.

HKCU

Abbreviation for **HKEY_CURRENT_USER**.

HKLM

Abbreviation for **HKEY_LOCAL_MACHINE**.

HKU

Abbreviation for **HKEY_USERS**.

HKR

Relative to the registry key most pertinent to the section in which this **AddReg** directive appears, such as the per-device "hardware" subkey in the registry **..\Enum\***enumeratorID\ device-instance-id* branch, the corresponding driver-specific "software" subkey under the registry **..Class\***SetupClassGUID\device-instance-id* branch, and so forth.

subkey

This optional value, formed either as a *%strkey%* token defined in a **Strings** section of the INF or as a registry path under the given *reg-root* (*key1\key2\key3...*), specifies one of the following:

- A new subkey to be added to the registry at the end of the given registry path
- An existing subkey in which the additional values specified in this entry will be written (possibly replacing the value of an existing named value entry of the given subkey)
- Both a new subkey to be added to the registry together with its initial value entry

value-entry-name

This optional value either names an existing value entry in the given (existing) *subkey* or creates the name of a new value entry to be added in the specified *subkey*, whether it already exists or is a new key to be added to the registry. This value can be expressed either as "*quoted string*" or as a *%strkey%* token that is defined in the INF's **Strings** section. (If this is omitted for a string-type value, the *value-entry-name* is the default "unnamed" value entry for this key in the Windows 2000 registry or the **Default** value entry in this key in the Windows 9x registry.)

The OS supports some system-defined special *value-entry-name* keywords. See the end of this **Comments** section for more information.

flag

This optional value, expressed as an ORed bitmask of system-defined flag values in hexadecimal notation, defines the data type for a value entry and/or controls the add-registry operation. Possible INF-specified bitmask value(s) for each of these system-defined flags include the following:

0x00000000 (= FLG_ADDREG_TYPE_SZ)

The given value entry and/or *value* is of type REG_SZ. Note that this is the default type for a specified value entry, so the *flags* value can be omitted from any *reg-root*= line in an add-registry section that operates on a value entry of this type.

0x00000001 (= FLG_ADDREG_BINVALUETYPE)

The given *value* is "raw" data. (This value is identical to the Windows 2000-specific FLG_ADDREG_TYPE_BINARY.)

0x00000002 (= FLG_ADDREG_NOCLOBBER)

Prevent a given *value* from replacing the value of an existing value entry.

0x00000004 (= FLG_ADDREG_DELVAL)

Delete the given *subkey* from the registry, or delete the specified *value-entry-name* from the specified registry *subkey*.

0x00000010 (= FLG_ADDREG_KEYONLY)

Create the given *subkey*, but ignore any supplied *value-entry-name* and/or *value*.

0x00000020 (= FLG_ADDREG_OVERWRITEONLY)

Reset to the supplied *value* only if the specified *value-entry-name* already exists in the given *subkey*.

0x00010000 (= FLG_ADDREG_TYPE_MULTI_SZ)

The given value entry and/or *value* is of the registry type REG_MULTI_SZ. This specification does not require any NUL terminator for a given string value.

0x00000008 (= FLG_ADDREG_APPEND)

Append a given *value* to that of an existing named value entry. This flag is valid only for value entries of type REG_MULTI_SZ.

0x00020000 (= FLG_ADDREG_TYPE_EXPAND_SZ)

The given value entry and/or *value* is of the registry type REG_EXPAND_SZ.

0x00010001 (= FLG_ADDREG_TYPE_DWORD)

The given value entry and/or *value* is of the registry type REG_DWORD.

0x00020001 (= FLG_ADDREG_TYPE_NONE)

The given value entry and/or *value* is of the Windows 2000 registry type REG_NONE.

value

This optionally specifies a new value for the specified *value-entry-name* to be added to the given registry key. Such a *value* can be a "replacement" value for an existing named value entry in an existing key, a value to be appended (*flag* value **0x00010008**) to an existing named REG_MULTI_SZ-type value entry in an existing key, a new value entry to be written into an existing key, or the initial value entry for a new *subkey* to be added to the registry.

The expression of such a *value* depends on the registry type specified for the *flag*, as follows:

- A registry string-type value can be expressed either as a "*quoted string*" or as a *%strkey%* token defined in a **Strings** section of the INF file. Such an INF-specified value need not include a NUL terminator at the end of each string.
- A registry numerical-type value can be expressed as a hexadecimal (using **0x** notation) or decimal number.

Note that the lower-order bit of the low word in a *flag* value distinguishes between character and binary data. Consequently, a Windows 95 installer interprets the extended Windows 2000 and Windows 98 registry data types as either REG_SZ or REG_BINARY.

To represent a number of a registry type other than one of the predefined REG_*XXX* types, specify a new type number in the high word of the *flag* ORed with FLG_ADDREG_BINVALUETYPE in its low word. The data for such a *value* must be specified in binary format as a sequence of bytes separated by commas. For example, to store 16 bytes of data of a new registry data type, such as 0x38, as a value entry, the add-registry section entry would be something like the following:

```
HKR,,MYValue,0x00380001,1,0,2,3,4,5,6,7,8,9,A,B,C,D,E,F
```

This technique can be used to define new registry types for numerical values, but not for values of type REG_EXPAND_SZ, REG_MULTI_SZ, REG_NONE, or REG_SZ.

Special *value-entry-name* Keywords

Windows 2000 defines special keywords for use in the HKR **AddReg** entries:

```
[ClassInstall32] | [ClassInstall32.ntx86] | [install-section-name.HW] | [install-section-name.nt*.HW]
...
AddReg = Xxx_AddReg
...

[Xxx_AddReg]
...
HKR,,DeviceCharacteristics,0x10001,characteristics
HKR,,DeviceType,0x10001,device-type
HKR,,Security,,security-descriptor-string
HKR,,UpperFilters,0x10000,service-name
HKR,,LowerFilters,0x10000,service-name
HKR,,Exclusive,0x10001,reserved          ; RESERVED. Do not use.
```

The special keywords are used as follows:

DeviceCharacteristics

A **DeviceCharacteristics** entry specifies characteristics for the device. The *characteristics* value is a numeric value that is the result of OR'ing one or more FILE_* file characteristics values as defined in *wdm.h* or *ntddk.h.*

Setup stores the specified device characteristics in a private location in the registry, separate from where Setup creates user-defined registry entries specified in other HKR **AddReg** entries.

A *characteristics* value of zero directs Setup to ignore the class-wide device characteristics, if any, that were specified in the associated class installer INF.

DeviceType

A **DeviceType** entry specifies a device type for the device. The *device-type* is the numeric value of a FILE_DEVICE_*XXX* constant defined in *wdm.h* or *ntddk.h.* The flag value of 0x10001 specifies that the *device-type* value is a REG_DWORD.

A class-installer INF should specify the device type that applies to all, or almost all, of the devices in the class. For example, if the devices in the class are of type FILE_DEVICE_CDROM, specify a *device-type* of 0x02. If a device INF specifies a **DeviceType,** it overrides the type set by the class installer, if any. If the class and/or device INF specifies a **DeviceType**, the PnP Manager applies that type to the PDO for the device.

Security

A **Security** entry specifies a security descriptor for the device. The *security-descriptor-string* is a string with tokens to indicate the DACL (**D:**) security component. See the Platform SDK documentation for more information on security descriptor strings. A class-installer INF can specify a security descriptor for a device class. A device INF can specify a security descriptor for an individual device, overriding the security for the class. If the class and/or device INF specifies a *security-descriptor-string*, the PnP Manager propagates the descriptor to all the device objects for a device, including the FDO, filter DOs, and the PDO.

UpperFilters

An **UpperFilters** entry specifies a PnP upper-filter driver. This entry in a *DDInstall*.**HW** section defines one or more device-specific upper-filter drivers. In a **ClassInstall32** section, this entry defines one or more class-wide upper-filter drivers.

LowerFilters

A **LowerFilters** entry specifies a PnP lower-filter driver. This entry in a *DDInstall*.**HW** section defines one or more device-specific lower-filter drivers. In a **ClassInstall32** section, this entry defines one or more class-wide lower-filter drivers.

Example

An **AddReg** directive referenced the (SCSI) Miniport_EventLog_AddReg section in this example under an INF-writer-defined section referenced by the **AddService** directive in a *DDInstall*.**Services** section of this INF. Registry entries for event-logging by all SCSI miniports are identical on Windows 2000 platforms.

```
[Miniport_EventLog_AddReg]
HKR,,EventMessageFile,0x00020000,"%%SystemRoot%%\System32\IoLogMsg.dll"
; double-quote delimiters in preceding entry prevent truncation
; if line wraps

HKR,,TypesSupported,0x00010001,7
```

See Also

AddInterface, **AddService**, **BitReg**, **ClassInstall32**, ***DDInstall***, ***DDInstall*.CoInstallers**, ***DDInstall*.HW**, ***DDInstall*.Interfaces**, ***DDInstall*.Services**, **DelReg**, **InterfaceInstall32**, **Strings**

INF DelReg Directive

[*DDInstall*] | [*DDInstall*.**HW**] | [*DDInstall*.**CoInstallers**] | [**ClassInstall32**] | [**ClassInstall32.ntx86**]

DelReg=*del-registry-section*[, *del-registry-section*] ...

A **DelReg** directive references one or more INF-writer-defined sections describing keys and/or value entries to be removed from the registry.

A **DelReg** directive can be specified under any of the sections shown in the formal syntax statement. This directive also can be specified under any of the following INF-writer-defined sections:

- A *service-install-section* or *event-log-install* section referenced by the **AddService** directive in a *DDInstall*.**Services** section
- An *add-interface-section* referenced by the **AddInterface** directive in a *DDInstall*.**Interfaces** section
- An *install-interface-section* referenced in an **InterfaceInstall32** section

Comments

In general, an INF should never attempt to delete subkeys or value entries within existing subkeys that were set up by system components or by the INFs for other devices. The purpose of a delete-registry section is to clean stale registry information from a previous installation using a new INF file supplied by the same provider.

Any given *del-registry-section* name must be unique to the INF file, but it can be referenced by **DelReg** directives in other sections of the same INF. Each INF-writer-defined section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*. For more information about how to use the system-defined **.nt** or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File*.

Each named section referenced by a **DelReg** directive has the following form:

[*del-registry-section*]

reg-root-string, *subkey*[, *value-entry-name*]
...

A *del-registry-section* can have any INF-writer-determined number of entries, each on a separate line.

DelReg-Referenced Section Entries

reg-root-string

Identifies the root of the registry tree for other values supplied in this entry. The value can be one of the following:

HKCR

Abbreviation for **HKEY_CLASSES_ROOT**.

HKCU

Abbreviation for **HKEY_CURRENT_USER**.

HKLM

Abbreviation for **HKEY_LOCAL_MACHINE**.

HKU

Abbreviation for **HKEY_USERS**.

HKR

Relative to the registry key most pertinent to the section in which this **DelReg** directive appears, such as the per-device "hardware" subkey in the registry **..\Enum\***enumeratorID\\ device-instance-id* branch, the corresponding driver-specific "software" subkey under the registry **..Class\***SetupClassGUID\\device-instance-id* branch, and so forth.

subkey

This value, formed either as a *%strkey%* token defined in a **Strings** section of the INF or as a registry path under the given *reg-root* (*key1\key2\key3...*), specifies one of the following:

- A subkey to be deleted from the registry at the end of the given registry path
- An existing subkey from which the given *value-entry-name* is to be deleted

value-entry-name

This value, formed either as a *%strkey%* token (defined in a **Strings** section) or as a "*quoted string*" identifies a named value entry to be removed from the given *subkey*. This value should be omitted if the given *subkey* is being removed from the registry.

Example

This example shows how the system-supplied COM/LPT ports class installer's INF removes stale Windows 2000-specific registry information about COM ports from the registry.

```
[ComPort.NT]
CopyFiles=ComPort.NT.Copy
AddReg=ComPort.AddReg, ComPort.NT.AddReg
 ... ; more directives omitted here

[ComPort.NT.HW]
DelReg=ComPort.NT.HW.DelReg

[ComPort.NT.Copy]
serial.sys
serenum.sys

[Comport.NT.AddReg]
HKR,,EnumPropPages32,,"MSPorts.dll,SerialPortPropPageProvider"

[ComPort.NT.HW.DelReg]
HKR,,UpperFilters
```

See Also

AddReg, **AddInterface**, **AddService**, **BitReg**, **ClassInstall32**, ***DDInstall***, ***DDInstall*.CoInstallers**, ***DDInstall*.HW**, ***DDInstall*.Services**, **InterfaceInstall32**, **Strings**

INF CopyFiles Directive

[*DDInstall*] | [*DDInstall*.**CoInstallers**] | **[ClassInstall32]** | **[ClassInstall32.ntx86]**

Copyfiles=@*filename* | *file-list-section*[**,***file-list-section*]...

A **CopyFiles** directive can do either of the following:

- Cause a single file to be copied from the source media to the default destination directory.
- Reference one or more INF-writer-defined sections in the INF that each specify a list of files to be copied from the source media to the destination.

A **CopyFiles** directive can be specified within any of the sections shown in the formal syntax statement. This directive also can be specified within any of the following INF-writer-defined sections:

- An *add-interface-section* referenced by the **AddInterface** directive in a *DDInstall*.**Interfaces** section
- An *install-interface-section* referenced in an **InterfaceInstall32** section

Comments

Any *file-list-section* name must be unique to the INF file, but it can be referenced by **CopyFiles**, **DelFiles**, or **RenFiles** directives elsewhere in the same INF. Such an INF-writer-defined section name must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*. For more information about how to use the system-defined **.nt** and **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide*.

Each named section referenced by a **CopyFiles** directive has one or more entries of the following form:

[*file-list-section*]
destination-file-name[,*source-file-name*][,*temporary-file-name*][,*flag*]
...

An INF-writer-defined *file-list-section* can have any number of entries, each on a separate line.

CopyFiles-Referenced Section Entries

destination-file-name

Specifies the name of the destination file. If no *source-file-name* is given, this specification is also the name of the source file.

source-file-name

Specifies the name of the source file. If the source and destination file names for the file copy operation are the same, *source-file-name* can be omitted.

temporary-file-name

Specifies the name of a temporary file to be created in the copy operation if a file of the same name on the destination is open or currently in use. Only used on Windows 9x platforms. Windows 2000 automatically generates temporary file names when necessary and renames the copied source files the next time the OS is started so this value is irrelevant in INFs for Windows 2000 device/driver installations.

flag

These optional flags, expressed in hexadecimal notation or as a decimal value in a section entry, can be used to control how (or whether) a particular source file is copied to the destination. One or more (ORed) values for the following system-defined flags can be specified, but some of these flags are mutually exclusive:

0x00000400 (COPYFLG_REPLACEONLY)

Copy the source file to the destination directory only if the file is already present in the destination directory.

0x00000800 (COPYFLG_NODECOMP)

Copy the source file to the destination directory without decompressing the source file if it is compressed.

0x00000008 (COPYFLG_FORCE_FILE_IN_USE)

Force file-in-use behavior: do not copy over an existing file of the same name if it is currently open. Instead, copy the given source file with a temporary name so that it can be renamed and used when the next reboot occurs.

0x00000010 (COPYFLG_NO_OVERWRITE)

Do not replace an existing file in the destination directory with a source file of the same name. This flag cannot be combined with any other flags.

0x00001000 (COPYFLG_REPLACE_BOOT_FILE)

This file is required by the system loader. The system will prompt the user to reboot the system.

0x00002000 (COPYFLG_NOPRUNE)

Do not delete this operation as a result of optimization.

For example, Setup might determine that the file copy operation is not necessary because the file already exists. However, the writer of the INF knows that the operation is required and directs Setup to override its optimization and perform the file operation.

0x00000020 (COPYFLG_NO_VERSION_DIALOG)

Do not write over a file in the destination directory with the source file if the existing file is newer than the source file.

This flag is irrelevant to digitally signed Windows 2000 INF files. If a driver package is digitally signed, Setup installs the package as a whole and does not selectively omit files in the package based on other versions already present on the machine.

0x00000004 (COPYFLG_NOVERSIONCHECK)

Ignore file versions and write over existing files in the destination directory. This flag and the next two are mutually exclusive. This flag is irrelevant to digitally signed Windows 2000 INF files.

0x00000040 (COPYFLG_OVERWRITE_OLDER_ONLY)

Copy the source file to the destination directory only if the file on the destination will be superceded by a newer version. This flag is irrelevant to digitally signed Windows 2000 INF files.

0x00000001 (COPYFLG_WARN_IF_SKIP)

Send a warning if the user elects to not copy a file. This flag and the next are mutually exclusive, and both are irrelevant to Windows 2000 INF files that are digitally signed.

0x00000002 (COPYFLG_NOSKIP)

Do not allow the user to skip copying a file. This flag is implied on Windows 2000 platforms if the driver package is signed.

The **DestinationDirs** section of an INF file controls the destination for all file-copy operations, whatever the section containing a particular **CopyFiles** directive, as follows:

- If a named section referenced by a **CopyFiles** directive has a corresponding entry in the **DestinationDirs** section of the same INF, that entry explicitly specifies the target destination directory into which all files listed in the named section will be copied. If the named section is not listed in the **DestinationDirs** section, Setup uses the **DefaultDestDir** in the INF.

- If a **CopyFiles** directive uses the @*filename* syntax, Setup uses the **DefaultDestDir** entry in the **DestinationDirs** section of the INF.

The INF also supplies path specification(s) to file(s) copied from source media in either of the following ways:

- In IHV/OEM-supplied INFs, by using the **SourceDisksNames** and, possibly, **SourceDisksFiles** sections of this INF to explicitly specify the full path to each named source file that is not in the root directory (or directories) on the distribution media

- In system-supplied INFs, by supplying one or more additional INF files, identified by name in the **LayoutFile** entry of the INF's **Version** section

Example

This example shows how the **SourceDisksNames**, **SourceDisksFiles**, and **DestinationDirs** sections specify the paths for copy-file (and delete-file) operations that occur in processing a simple device-driver INF. (The same INF was also used previously as examples of **Version**, **SourceDisksNames**, and **SourceDisksFiles** sections.)

```
[SourceDisksNames]
1 = %Floppy_Description%,,,\Win98 ; path to Win98 source files
2 = %Floppy_Description%,,,\WinNT

[SourceDisksFiles]
aha154x.mpd = 1,, ; on distribution disk 1, in subdir \win98

[SourceDisksFiles.x86]
aha154x.sys = 2,\x86 ; on distribution disk 2, in subdir \WinNT\x86

[DestinationDirs]
ASPIDEV = 11  ; Win98-specific del-file section
              ; delete existing file(s) from DIRID_SYSTEM
DefaultDestDir = 12  ; DIRID_DRIVERS
                     ; == \System32\Drivers on Windows NT platforms
                     ; == \System\IoSubsys on Win9x platforms

; ... Manufacturer and Models sections omitted here

; Win9x-specific DDInstall, given [AHA154X.NTx86] in this INF
[AHA154X]
CopyFiles=@AHA154x.MPD
DelFiles=ASPIDEV ; defines a delete-files section not shown here
; ... some other directives and sections omitted here

[AHA154X.NTx86]
CopyFiles=@AHA154x.SYS
; ... some other directives and sections omitted here
; ...
```

This example shows how a **CopyFiles** directive can be used in a *DDInstall*.**CoInstallers** section of an INF for a device driver that provides two device-specific coinstallers to supplement the INF processing of the system device-type-specific class installer.

```
[DestinationDirs]
XxDev_Coinstallers_CopyFiles = 11  ; DIRID_SYSTEM
; ... other file-list entries and DefaultDestDirs omitted here

; ... Manufacturer, Models, and DDInstall sections omitted here
```

```
[XxDev_Install.CoInstallers]
CopyFiles=XxDev_Coinstallers_CopyFiles
; ... AddReg omitted here

[XxDev_Coinstallers_CopyFiles]
XxPreInst.dll   ; dev-specific coinstaller run before class installer
XxPostInst.dll  ; run after class installer (post processing)
```

As the preceding example suggests, the names of new device-specific coinstallers can be constructed from the name of the provider (shown here as *Xx*) and the intended use for each such coinstaller DLL (shown here as *PreInst* and *PostInst*).

See Also

AddInterface, **ClassInstall32**, ***DDInstall***, ***DDInstall*.CoInstallers**, ***DDInstall*.Interfaces**, **DelFiles**, **DestinationDirs**, **InterfaceInstall32**, **RenFiles**, **SourceDisksFiles**, **SourceDisks-Names**, **Version**

INF DelFiles Directive

[*DDInstall*] | [*DDInstall*.**CoInstallers**] | **[ClassInstall32]** | **[ClassInstall32.ntx86]**

Delfiles=*file-list-section*[**,** *file-list-section*]...

A **DelFiles** directive references an INF-writer-defined section elsewhere in the INF file, causing that list of files to be deleted in the context of operations on the section in which the referring **DelFiles** directive is specified.

A **DelFiles** directive can be specified within any of the sections shown in the formal syntax statement. This directive also can be specified within any of the following INF-writer-defined sections:

- An *add-interface-section* referenced by the **AddInterface** directive in a ***DDInstall*.Interfaces** section
- An *install-interface-section* referenced in an **InterfaceInstall32** section

Comments

Typically, this directive is used only in an "upgrade" INF file to delete obsolete (driver) files for a previous installation of the same device(s) from the target machine. However, the replacement driver(s) should be tested with any application(s) and/or other drivers that might depend on the previously installed driver(s) before such an upgrade INF deletes stale driver file(s).

Any *file-list-section* name must be unique to the INF file, but it can be referenced by **Copy-Files**, **DelFiles**, or **RenFiles** directives elsewhere in the same INF. Such an INF-writer-defined section name must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*. For more information about how to use the system-defined **.nt** and/or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide*.

Each named section referenced by a **DelFiles** directive has one or more entries of the following form:

[*file-list-section*]

destination-file-name[,,,*flag*]
...

A *file-list-section* can have any number of entries, each on a separate line.

DelFiles-Referenced Section Entries

destination-file-name

Specifies the name of the file to be deleted from the destination.

Do not specify a file that is listed in a **CopyFiles** directive. If a file is listed in both a **Copy-Files**-referenced and a **DelFiles**-referenced section, and the file is currently present on the system with a valid signature, the OS might optimize away the copy operation but perform the delete operation. This is very likely *not* what the INF writer intended.

flag

This optional value can be one of the following, expressed in hexadecimal notation as shown here or as a decimal value:

0x00000001 (DELFLG_IN_USE)

Delete the named file, possibly after it has been used during the installation process, either on a Windows 2000 or Windows 9x machine.

Setting this flag value in an INF queues the file-deletion operation until the system has restarted if the given file cannot be deleted because it is in use while this INF is being processed. Otherwise, such a file will not be deleted.

0x00010000 (DELFLG_IN_USE1)

This flag is a high-word version of the DELFLG_IN_USE flag, and it has the same purpose and effect. This flag should be used in Windows 2000-only INF files.

Setting this flag value in an INF prevents conflicts with the COPYFLG_WARN_IF_SKIP flag in an INF with both **DelFiles** and **CopyFiles** directives that reference the same *file-list-section*.

The **DestinationDirs** section of the INF file controls the destination for all file-deletion operations, whatever the section containing a particular **DelFiles** directive, as follows:

- If a named section referenced by a **DelFiles** directive has a corresponding entry in the **DestinationDirs** section of the same INF, that entry explicitly specifies the target destination directory from which all files listed in the named section will be deleted. If the named section is not listed in the **DestinationDirs** section, Setup uses the **DefaultDestDir** entry in the INF.

Example

This example shows how the **DestinationDirs** section specifies the path for a delete-file operation that occurs in processing a simple device-driver INF. (The same INF was also used previously as an example of the **CopyFiles** directive.)

```
[DestinationDirs]
ASPIDEV = 11  ; name of Win9x-specific delete-file section
              ; delete existing file(s) from DIRID_SYSTEM
DefaultDestDir = 12  ; DIRID_DRIVERS

; ...

[AHA154X] ; Win9x-specific DDInstall section
CopyFiles=@AHA154x.MPD
DelFiles=ASPIDEV ; defines delete-files section name
; ... some other directives and sections omitted here

[ASPIDEV]
VASPID.VXD ; name of file to be deleted, if it exists on target
; ...
```

See Also

AddInterface, **ClassInstall32**, **CopyFiles**, ***DDInstall***, ***DDInstall*.CoInstallers**, **DestinationDirs**, **InterfaceInstall32**, **RenFiles**

INF RenFiles Directive

[*DDInstall*] | [*DDInstall*.**CoInstallers**] | [**ClassInstall32**] | [**ClassInstall32.ntx86**]

Renfiles=*file-list-section*[**,** *file-list-section*]...

A **RenFiles** directive references an INF-writer-defined section elsewhere in the INF file, causing that list of files to be renamed in the context of operations on the section in which the referring **RenFiles** directive is specified.

A **RenFiles** directive can be specified within any of the sections shown in the formal syntax statement. This directive also can be specified within any of the following INF-writer-defined sections:

- An *add-interface-section* referenced by the **AddInterface** directive in a *DDInstall.* **Interfaces** section
- An *install-interface-section* referenced in an **InterfaceInstall32** section

Comments

Typically, this directive is used only in an "upgrade" INF file to preserve files that were installed on the target machine in a previous installation of the same device(s) if some new files supplied on the source media have the same names.

Any *file-list-section* name must be unique to the INF file, but it can be referenced by **CopyFiles**, **DelFiles**, or **RenFiles** directives elsewhere in the same INF. Such an INF-writer-defined section name must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*. For more information about how to use the system-defined **.nt** and/or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File*.

Each named section referenced by a **RenFiles** directive has one or more entries of the following form:

[*file-list-section*]

new-dest-file-name,old-source-file-name
...

A *file-list-section* can have any number of entries, each on a separate line.

RenFiles-Referenced Section Entries

new-dest-file-name

Specifies the new name to be given to the file on the destination.

old-source-file-name

Specifies the old name of the file.

The **DestinationDirs** section of the INF file controls the destination for all file-rename operations, whatever the section containing a particular **RenFiles** directive, as follows:

- If a named section referenced by a **RenFiles** directive has a corresponding entry in the **DestinationDirs** section in the same INF, that entry explicitly specifies the target destination directory in which all files listed in the named section will be renamed on the destination before these source files are copied. If the section is not listed in the **DestinationDirs** section, Setup uses the **DefaultDestDir** entry in the INF.

Example

This example shows a section referenced by a **RenFiles** directive.

```
[RenameOldFilesSec]
devfile41.sav, devfile41.sys
```

See Also

AddInterface, **ClassInstall32**, **CopyFiles**, ***DDInstall***, ***DDInstall*.CoInstallers**, **DelFiles**, **DestinationDirs**, **InterfaceInstall32**, **SourceDisksFiles**, **SourceDisksNames**, **Version**

INF AddService Directive

[*DDInstall*.**Services**]

AddService=*ServiceName*,[*flags*],*service-install-section*[,
    event-log-install-section[,[*EventLogType*][,*EventName*]]]
...

An **AddService** directive is used in a *DDInstall*.**Services** section to control how (and when) the services of particular Windows 2000 device(s)' driver(s) are loaded, any dependencies on other underlying legacy drivers or services, and so forth. Optionally, this directive sets up event-logging services by the device/driver(s) as well.

This directive is irrelevant to exclusively Windows 9x INF files. It also is not used in Windows 2000 INF files that install devices, such as modems or display monitors, that do not install any drivers.

Entry Values

ServiceName

Specifies the name of the service to be installed. For a device, this value is usually a generic name for its driver, such as "sermouse," or some such name.

flags

Specifies one or more (ORed) of the following system-defined flags, expressed as a hexadecimal value:

0x00000002 (SPSVCINST_ASSOCSERVICE)

Mark the named service as the PnP function driver (or legacy driver) for the particular device being installed by this INF file. Such a device might be any one of the device(s) or device-compatible models listed in an entry of the *Models* section that referenced this *DDInstall* section. Do not specify this flag on filter drivers.

0x00000008 (SPSVCINST_NOCLOBBER_DISPLAYNAME)

Do not overwrite the given service's (optional) friendly name if this service already exists in the system.

0x00000100 (SPSVCINST_NOCLOBBER_DESCRIPTION)

Do not overwrite the given service's (optional) description if this service already exists in the system.

0x00000010 (SPSVCINST_NOCLOBBER_STARTTYPE)

Do not overwrite the given service's start type if this named service already exists in the system.

0x00000020 (SPSVCINST_NOCLOBBER_ERRORCONTROL)

Do not overwrite the given service's error-control value if this named service already exists in the system.

0x00000040 (SPSVCINST_NOCLOBBER_LOADORDERGROUP)

Do not overwrite the given service's load-order-group value if this named service already exists in the system. INFs that install exclusively PnP devices and devices with WDM drivers should not set this flag.

0x00000080 (SPSVCINST_NOCLOBBER_DEPENDENCIES)

Do not overwrite the given service's dependencies list if this named service already exists in the system. INFs that install exclusively PnP devices and devices with WDM drivers should not set this flag.

0x00000001 (SPSVCINST_TAGTOFRONT)

Move the named service's tag to the front of its group order list, thereby ensuring that it is loaded first within that group (unless a subsequently installed device with this INF specification displaces it). INFs that install exclusively PnP devices and devices with WDM drivers should not set this flag.

service-install-section

References an INF-writer-defined section that contains information for installing the named service for this device (or devices).

event-log-install-section

Optionally references an INF-writer-defined section in which event-logging services for this device (or devices) are set up.

EventLogType

Optionally specifies one of **System**, **Security**, or **Application**. If omitted, this defaults to **System**, which is almost always the appropriate value for the installation of device drivers. For example, an INF would specify **Security** only if the to-be-installed driver provides its own security support.

EventName

Optionally specifies a name to use for the event log. If omitted, this defaults to the given *ServiceName*.

Comments

The system-defined and case-insensitive extensions can be inserted into a *DDInstall.***Services** section containing an **AddService** directive in cross-OS and/or cross-platform INF files to specify platform-specific or OS-specific installations. For more information about how to use the system-defined **.nt** and/or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide.*

Each INF-writer-created section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files.*

An **AddService** directive must reference a named *service-install-section* elsewhere in the INF file. Each such named section has the following form:

[*service-install-section*]

[**DisplayName=***name*]
[**Description=***description-string*]
ServiceType=*type-code*
StartType=*start-code*
ErrorControl=*error-control-level*
ServiceBinary=*path-to-service*
[**StartName=***driver-object-name*]
[**AddReg=***add-registry-section*[**,** *add-registry-section*] ...]
[**DelReg=***del-registry-section*[**,** *del-registry-section*] ...]
[**BitReg=***bit-registry-section*[**,***bit-registry-section*] ...]
[**LoadOrderGroup=***load-order-group-name*]
[**Dependencies=***depend-on-item-name*[**,***depend-on-item-name*]...]

Any *service-install-section* must have at least the **ServiceType**, **StartType**, **ErrorControl**, and **ServiceBinary** entries as shown here, but the remaining entries are optional.

Service-Install Section Entries and Values

DisplayName=name

Specifies a friendly name for the service/driver, usually, for ease of localization, expressed as a *%strkey%* token defined in a **Strings** section of the INF file.

Description=description-string

Optionally specifies a string that describes the service, usually expressed as a *%strkey%* token defined in a **Strings** section of the INF file.

This string gives the user more information about the service than the **DisplayName**. For example, the **DisplayName** might be something like "DHCP Client" and the **Description** might be something like "Manages network configuration by registering and updating IP addresses and DNS names".

The *description-string* should be long enough to be descriptive but not so long as to be awkward. If a *description-string* contains any *%strkey%* tokens, each token can represent a maximum of 511 characters. The total string, after any string token substitutions, should be no longer than 1024 characters. Most strings will be shorter.

ServiceType=type-code

In any INF that installs support for one or more devices, the *type-code* specification must be **1** or **0x00000001**, that is, equivalent to SERVICE_KERNEL_DRIVER.

However, a device-dedicated application supplied when installing the device would be classified as either of SERVICE_WIN32_OWN_PROCESS (specified as **10** or **0x00000010** in the INF) or SERVICE_WIN32_SHARE_PROCESS (specified as **20** or **0x00000020** in the INF). A highest level network driver, such as a redirector, or a file system would be classified as SERVICE_KERNEL_FILE_SYSTEM_DRIVER by specifying **2** or **0x00000002** in its INF.

StartType=start-code

Specifies when to start the driver as one of the following numerical values, expressed either in decimal or, as shown here, in hexadecimal notation.

0x0 (SERVICE_BOOT_START)

Indicates a driver started by the operating system loader.

This value should be used *only* for drivers of devices required for loading the OS.

0x1 (SERVICE_SYSTEM_START)

Indicates a driver started during OS initialization.

This value should be used by Windows 2000 PnP drivers that do device detection during initialization but are not required to load the system.

For example, a PnP driver that also can detect a legacy device should specify this value in its INF so that its **DriverEntry** routine will be called to find the legacy device, even if that device cannot be enumerated by the PnP Manager.

0x2 (SERVICE_AUTO_START)

Indicates a driver started by the Service Control Manager during system startup.

This value should never be used in the INF files for WDM or Windows 2000 PnP device drivers.

0x3 (SERVICE_DEMAND_START)

Indicates a driver started on demand, either by the PnP Manager when the corresponding device is enumerated or possibly by the Service Control Manager in response to an explicit user demand for a nonPnP device.

This value should be used in the INF files for all WDM drivers of devices that are not required to load the system and for all Windows 2000 PnP device drivers that are neither required to load the system nor engaged in device detection.

0x4 (SERVICE_DISABLED)

Indicates a driver that cannot be started.

This value can be used to temporarily disable the driver services for a device, but a device/driver cannot be installed if this value is specified in the service-install section of its INF file.

ErrorControl=error-control-level

Specifies the level of error control as one of the following numerical values, expressed either in decimal or, as shown here, in hexadecimal notation.

0x0 (SERVICE_ERROR_IGNORE)

If the driver fails to load or initialize, proceed with system startup and do not display a warning to the user.

0x1 (SERVICE_ERROR_NORMAL)

If the driver fails to load or initialize its device, system startup should proceed but display a warning to the user.

0x2 (SERVICE_ERROR_SEVERE)

If the driver fails to load, system startup should switch to the registry's **LastKnown-Good** control set and continue system startup, even if the driver again indicates a loading or device/driver initialization error.

0x3 (SERVICE_ERROR_CRITICAL)

If the driver fails to load and system startup is not using the Windows 2000 registry's **Last-KnownGood** control set, switch to **LastKnownGood** and try again. If startup still fails when using **LastKnownGood**, run a bug-check routine. (*Only* devices/drivers necessary for the system to boot specify this value in their INF files.)

ServiceBinary=path-to-service

Specifies the path to the binary for the service, expressed as *%dirid%\filename*.

The *dirid* number is one of the system-defined directory identifiers already described in the reference for the **DestinationDirs** section. The given *filename* specifies a file already transferred (see *INF **CopyFiles** Directive*) from the source distribution media to that directory on the target machine.

StartName=driver-object-name

This optional entry specifies the name of the driver object representing this device/driver. If *type-code* specifies **1** or **2** (SERVICE_KERNEL_DRIVER or SERVICE_FILE_SYSTEM_DRIVER), this name is the Windows 2000 driver object name that the I/O Manager uses to load the driver.

AddReg=add-registry-section[, add-registry-section] ...

References one or more INF-writer-defined *add-registry-section*s in which any registry information pertinent to the newly installed service(s) is set up. This directive is rarely used in a *service-install-section*. An **HKR** specification in such an add-registry section designates the **HKLM\System\CurrentControlSet\Services\***ServiceName* registry key.

DelReg=del-registry-section[, del-registry-section] ...

References one or more INF-writer-defined *del-registry-section*s in which pertinent registry information for an already installed service(s) is removed. An **HKR** specification in such a del-registry section designates the **HKLM\System\CurrentControlSet\Services\***ServiceName* registry key. This directive is almost never used in a *service-install-section*, but it might be used in an INF that "updates" the registry for a previous installation of the same device/driver services.

BitReg=bit-registry-section[,bit-registry-section] ...

Is valid in a *service-install-section* but almost never used. An **HKR** specification in such a bit-registry section also designates the **HKLM\System\CurrentControlSet\Services\** *ServiceName* registry key.

LoadOrderGroup=load-order-group-name

This optional entry identifies the Windows 2000 load order group of which this driver is a member. It can be one of the "standard" load order groups, such as **SCSI class** or **NDIS**, defined by previous versions of Windows NT®.

In general, this entry is unnecessary for devices with WDM drivers or for exclusively PnP devices, unless there are legacy dependencies on such a group. However, this entry can be useful if device detection is supported by loading a group of drivers in a particular order.

Dependencies=depend-on-item-name[,depend-on-item-name]...

Each *depend-on-item-name* in a dependencies list specifies the name of a service or load-order group on which the device/driver depends.

If the *depend-on-item-name* specifies a service, the service that must be running before this driver is started. For example, the INF for the system-supplied Win32 TCP/IP print services depends on the support of the underlying (kernel-mode) TCP/IP transport stack. Consequently, the INF for the TCP/IP print services specifies this entry as **Dependencies =TCPIP**.

A *depend-on-item-name* can specify a load order group on which this device/driver depends. Such a driver is started only if at least one member of the specified group has been started. Preceed the group name with a plus sign (+). For example, the system RAS services INF might have an entry like **Dependencies = +NetBIOSGroup,RpcSS**, which lists both a load-order group and a service.

The OS loads drivers according to the service-install section *start-code* specifications of their INFs, as follows:

1. During the system boot start phase, the OS loads all **0x0** (SERVICE_BOOT_START) drivers.
2. During the system start phase, the OS first loads all WDM and Windows 2000 PnP drivers for which the PnP Manager finds devnodes in the registry **..\Enum** tree (whether their INFs say **0x01** for SERVICE_SYSTEM_START or **0x03** for SERVICE_DEMAND_START).Then the OS loads all remaining SERVICE_SYSTEM_START drivers.
3. During the system auto-start phase, the OS loads all remaining SERVICE_AUTO_START drivers.

For more information, see *How Does Setup Select a Driver For a Device?* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide.*

Registering for Event Logging

An **AddService** directive also can reference an INF-writer-defined *event-log-install-section* elsewhere in the INF file.

Each such named section referenced by an **AddService** directive has the following form:

[*event-log-install-section*]

AddReg=*add-registry-section*[, *add-registry-section*]...
[**DelReg=***del-registry-section*[, *del-registry-section*]...]
[**BitReg=***bit-registry-section*[,*bit-registry-section*]...]
...

For a typical device/driver INF, the event-log-install section uses only the **AddReg** directive to reference a single INF-writer-defined section that sets up an event-logging message file for the driver in the registry. An **HKR** specification in such an add-registry section designates the **HKLM\System\CurrentControlSet\Services\EventLog\***EventLogType\EventName* registry key. This event-logging add-registry section has the following general form:

[*drivername_EventLog_AddReg*]
HKR,,EventMessageFile,0x00020000,"*path*\IoLogMsg.dll;*path\driver.sys*"
HKR,,TypesSupported,0x0001001,7

In particular, such a section adds two value entries in the registry subkey created for the device/driver, as follows:

- The value entry named **EventMessageFile** is of type REG_EXPAND_SZ, as specified by the FLG_ADDREG_TYPE_EXPAND_SZ value **0x0002000**. Its given value, enclosed in double quotes (") associates the system-supplied **IoLogMsg.dll** (but it could associate another logging DLL) with the driver binary. Usually, the Windows 2000 *path*s to each of these files is expressed as follows:

  %%SystemRoot%%\System32\IoLogMsg.dll, and %%SystemRoot%%\System32\drivers\driver.sys, unless the driver binary was copied into another subdirectory on the target machine, as specified in the **DestinationDirs** section or in a section referenced by a **CopyFiles** directive elsewhere in the INF file.

- The value entry named **TypesSupported** is of type REG_DWORD, as specified by the FLG_ADDREG_TYPE_DWORD value **0x00010000**. For drivers, this value should be **7** (equivalent to the OR of EVENTLOG_SUCCESS, EVENTLOG_ERROR_TYPE, EVENTLOG_WARNING_TYPE, and EVENTLOG_INFORMATION_TYPE, without setting the EVENTLOG_AUDIT_*XXX* bits).

However, an event-log-install section also can use the **DelReg** directive to reference a section in which a previously installed event-log message file is removed from the registry by explicitly deleting the existing **EventMessageFile** and **TypesSupported** value entries for a driver binary that is being superceded by a newly installed driver. (See also the reference for the **DelService** directive.) While a **BitReg** directive also is valid within an INF-writer-defined *event-log-install-section*, it is almost never used, because the standard value entries for device driver event logging are not bitmasks.

Example

This example shows the service-install and event-log-install sections referenced by the **AddService** directive as already shown earlier in the example for *DDInstall*.**Services**.

```
[sermouse_Service_Inst]
DisplayName    = %sermouse.SvcDesc%
ServiceType    = 1                    ; = SERVICE_KERNEL_DRIVER
StartType      = 3                    ; = SERVICE_DEMAND_START
ErrorControl   = 1                    ; = SERVICE_ERROR_NORMAL
ServiceBinary  = %12%\sermouse.sys
LoadOrderGroup = Pointer Port

[sermouse_EventLog_Inst]
AddReg = sermouse_EventLog_AddReg

[sermouse_EventLog_AddReg]
HKR,,EventMessageFile,0x00020000,"%%SystemRoot%%\System32\IoLogMsg.dll;
       %%SystemRoot%%\System32\drivers\sermouse.sys"
;
; Preceding entry on single line in INF file. Enclosing quotes
; prevent the semicolon from being interpreted as a comment.
;
HKR,,TypesSupported,0x00010001,7

[mouclass_Service_Inst]
DisplayName    = %mouclass.SvcDesc%
ServiceType    = 1                    ; = SERVICE_KERNEL_DRIVER
StartType      = 1                    ; = SERVICE_SYSTEM_START
ErrorControl   = 1                    ; = SERVICE_ERROR_NORMAL
ServiceBinary  = %12%\mouclass.sys
LoadOrderGroup = Pointer Class

[mouclass_EventLog_Inst]
AddReg = mouclass_EventLog_AddReg

[mouclass_EventLog_AddReg]
HKR,,EventMessageFile,0x0002000,"%%SystemRoot%%\System32\IoLogMsg.dll;
       %%SystemRoot%%\System32\drivers\mouclass.sys"
HKR,,TypesSupported,0x00010001,7
; ...
[Strings]
; ...
sermouse.SvcDesc = "Serial Mouse Driver"
mouclass.SvcDesc = "Mouse Class Driver"
```

The example in the reference for the *DDInstall*.**HW** section, described earlier, also shows some service-install sections referenced by the **AddService** directive to set up PnP upper-filter drivers.

See Also

AddReg, **BitReg**, **CopyFiles**, ***DDInstall*.HW**, ***DDInstall*.Services**, **DelReg**, **DelService**, **DestinationDirs**, **Strings**

INF DelService Directive

[*DDInstall*.**Services**]

DelService=*ServiceName*[,[*flags*]][,[*EventLogType*][,*EventName*]]]
...

A **DelService** directive is used in a *DDInstall*.**Services** section to remove one or more previously installed device/driver service(s) from the target machine.

Entry Values

ServiceName

Specifies the name of the service to be removed.

For a device, this value is usually a generic name for its driver, such as "sermouse," or some such name.

flags

This optional value is specifies one or more of the following flags:

SPSVCINST_DELETEEVENTLOGENTRY

An event-log entry (or entries) associated with the given *ServiceName* should also be removed from the system.

SPSVCINST_STOPSERVICE

Stop the service before deleting it.

EventLogType

Optionally specifies one of **System**, **Security**, or **Application**. This can be omitted if the event log to be removed is of type **System**.

EventName

Optionally specifies the name for the event log. This can be omitted if it is identical to the given *ServiceName*.

Comments

This directive is almost never used, except possibly in an INF file that upgrades a previous installation of the same devices/models listed in the per-manufacturer per-*Models* section that referenced this *DDInstall* section and, in the upgrade process, supercedes previously installed services.

However, by default, event-log information supplied by a particular device driver is not removed from the system on deinstallation, unless the INF for the device/driver explicitly requests the removal (*flags* or *EventName*) of the event log along with the removal of the driver services.

See Also

AddService, ***DDInstall*.Services**, **DelReg**

INF AddInterface Directive

[*DDInstall*.**Interfaces]**

AddInterface={*InterfaceGUID***}** [, [*reference string*] [,[*add-interface-section*] [,flags]]]

One or more **AddInterface** directives can be specified to install support for device interfaces exported to higher level components, such as other drivers or applications, and to reference the add-interface section typically used for setting up registry information about the device/driver support of each such device interface.

An exported device interface can be one of the system-defined device interfaces, such as those defined by kernel streaming, or even a new device interface class to be exported for any subsequently installed device/driver that can use it.

Values

InterfaceGUID

Either specifies a new GUID value generated for this device/driver or, more commonly, specifies one of the system-defined GUID values of a device interface supported by the to-be-installed driver or component. This can be expressed as an explicit GUID value of the form {*nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn*} or as a *%strkey%* token defined to "{*nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn*}" in a **Strings** section of the INF file.

For more information about how to create a GUID, see the Platform SDK documentation. For the system-defined interface class GUIDS, see the appropriate header, such as *ks.h* for the kernel-streaming interface GUIDs.

reference string

This optional value associated with the given interface can be expressed either as a "*quoted string*" or as a *%strkey%* token defined in a **Strings** section of the INF file.

PnP function and filter drivers usually omit this value from the **AddInterface=** entries in their INF files, but such a *reference string* can be used as a placeholder for software devices that are created on demand. That is, the same *InterfaceGUID* value can be specified in INF entries with two or more unique *reference string*s. Because the I/O Manager passes such a *reference string* value as a path component of the interface's name whenever it is opened, the installed driver thus can discriminate between interfaces of the same class for a single device.

add-interface-section

References the name of a section elsewhere in the INF file, which typically has an **AddReg** directive to set up the registry entries exporting the driver's support of this device interface class.

flags

If specified, must be zero.

Comments

The system-defined and case-insensitive extensions can be inserted into a *DDInstall.* **Interfaces** section containing an **AddInterface** directive in cross-OS and/or cross-platform INF files to specify platform-specific or OS-specific interface installations. For more information about how to use the system-defined **.nt** and/or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File.*

If a given {*InterfaceGUID*} is not installed already, the system setup code installs that device-interface class in the system. Any INF that installs such a new device-interface class typically also has an **[InterfaceInstall32]** section containing the specified {*InterfaceGUID*} as an entry and referencing an INF-writer-defined *interface-install-section* that sets up the new device-interface-specific installation operations for that device-interface class.

To enable a device interface for runtime use by higher level components, the driver must make a call **IoSetDeviceInterfaceState** with each {*InterfaceGUID*} value identifying a device interface class that the driver supports on the underlying device. As an alternative to registering its support for a device interface in its INF, a device driver can call **IoRegisterDeviceInterface** before making its initial call to **IoSetDeviceInterfaceState**. Usually, a PnP function or filter driver makes this call from its AddDevice routine.

Each **AddInterface** directive in a *DDInstall*.**Interfaces** section can reference an INF-writer-defined *add-interface-section* elsewhere in the INF file. Each INF-writer-defined section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*.

An add-interface section referenced by the **AddInterface** directive has the following form:

[*add-interface-section*]

AddReg=*add-registry-section*[, *add-registry-section*]...
[**DelReg=***del-registry-section*[, *del-registry-section*] ...]
[**BitReg=***bit-registry-section*[,*bit-registry-section*] ...]
[**CopyFiles=**@*filename* | *file-list-section*[,*file-list-section*]...]
[**DelFiles=***file-list-section*[,*file-list-section*]...]
[**RenFiles=***file-list-section*[,*file-list-section*]...]
[**UpdateInis=***update-ini-section*[, *update-ini-section*] ...]
[**UpdateIniFields=***update-inifields-section*[, *update-inifields-section*] ...]
[**Ini2Reg=***ini-to-registry-section*[, *ini-to-registry-section*] ...]

Typically, an add-interface section contains only an **AddReg** directive that, in turn, refernces a single add-registry section. The add-registry section is used to store information in the registry about the interface(s) supported by the device driver for subsequent use by still higher level drivers and applications.

An add-registry section referenced within such an add-interface section is both device/driver-specific and interface-specific in nature. However, it must have a value entry defining a friendly name for the exported device interface so that still higher level components can refer to that interface by its friendly name in the user interface. An **HKR** specified in such an add-registry section designates the user-mode-accessible device-interface subkey of the **..DeviceClasses\\{***InterfaceGUID***}** branch. In the Windows 2000 registry, **DeviceClasses** is a subkey of the **..CurrentControlSet\Control** key.

Example

This example shows some of the expansion of the *DDInstall***.Interfaces** section for a particular audio device that supports system-defined kernel-streaming interfaces.

```
; ...
[ESS6881.Device.Interfaces]
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_Wave%,ESSAud.Interface.Wave
AddInterface=%KSCATEGORY_RENDER%,%KSNAME_Wave%,ESSAud.Interface.Wave
AddInterface=%KSCATEGORY_CAPTURE%,%KSNAME_Wave%,ESSAud.Interface.Wave
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_Topology%,\
ESSAud.Interface.Topology
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_UART%,WDM.Interface.UART
AddInterface=%KSCATEGORY_RENDER%,%KSNAME_UART%,WDM.Interface.UART
AddInterface=%KSCATEGORY_CAPTURE%,%KSNAME_UART%,WDM.Interface.UART
AddInterface=%KSCATEGORY_AUDIO%,%KSNAME_FMSynth%,WDM.Interface.FMSynth
AddInterface=%KSCATEGORY_RENDER%,%KSNAME_FMSynth%,\
WDM.Interface.FMSynth
```

```
[ESSAud.Interface.Wave]
AddReg=ESSAud.Interface.Wave.AddReg

[ESSAud.Interface.Wave.AddReg]
HKR,,CLSID,,%Proxy.CLSID%
HKR,,FriendlyName,,%ESSAud.Wave.szPname%
; ...
[WDM.Interface.UART]
AddReg=WDM.Interface.UART.AddReg

[WDM.Interface.UART.AddReg]
HKR,,CLSID,,%Proxy.CLSID%
HKR,,FriendlyName,,%WDM.UART.szPname%
; ...
[Strings]
KSCATEGORY_AUDIO="{6994ad04-93ef-11d0-a3cc-00a0c9223196}"
KSCATEGORY_RENDER="{65e8773e-8f56-11d0-a3b9-00a0c9223196}"
KSCATEGORY_CAPTURE="{65e8773d-8f56-11d0-a3b9-00a0c9223196}"
; ...
KSNAME_Wave="Wave"
KSNAME_UART="UART"
; ...
Proxy.CLSID="{17cca71b-ecd7-11d0-b908-00a0c9223196}"
; ...
ESSAud.Wave.szPname="ESS AudioDrive"
; ...
```

See Also

AddReg, **BitReg**, **CopyFiles**, ***DDInstall*.Interfaces**, **DelFiles**, **DelReg**, **Ini2Reg**, **InterfaceInstall32**, **IoRegisterDeviceInterface**, **IoSetDeviceInterfaceState**, **RenFiles**, **UpdateIniFields**, **UpdateInis**

INF BitReg Directive

[*DDInstall*] | [*DDInstall*.**HW**] | [*DDInstall*.**CoInstallers**] | [**ClassInstall32**] | [**ClassInstall32.ntx86**]

BitReg=*bit-registry-section*[,*bit-registry-section*]...

A **BitReg** directive references one or more INF-writer-defined sections used to set or clear bits within an existing REG_BINARY-type value entry in the registry. However, this directive is very rarely used in device/driver INF files.

A **BitReg** directive can be specified under any of the sections shown in the formal syntax statement. This directive also can be specified under any of the following INF-writer-defined sections:

- A *service-install-section* or *event-log-install* section referenced by the **AddService** directive in a *DDInstall*.**Services** section
- An *add-interface-section* referenced by the **AddInterface** directive in a *DDInstall*.**Interfaces** section
- An *install-interface-section* referenced in an **InterfaceInstall32** section

Comments

A given *bit-registy-section* name must be unique to the INF file, but it can be referenced by **BitReg** directives in other sections of the same INF. Each INF-writer-created section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*. For more information about how to use the system-defined **.nt** and/or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide*.

Each named section referenced by an **BitReg** directive has the following form:

[*bit-registry-section*]

reg-root, [*subkey*], *value-entry-name*, [*flag*], *byte-mask*, *byte-to-modify*
...

A *bit-registry-section* can have any INF-writer-determined number of entries, each on a separate line.

BitReg-Referenced Section Entries

reg-root

Identifies the root of the registry tree for other values supplied in this entry. The value can be one of the following:

HKCR

Abbreviation for **HKEY_CLASSES_ROOT**.

HKCU

Abbreviation for **HKEY_CURRENT_USER**.

HKLM

Abbreviation for **HKEY_LOCAL_MACHINE**.

HKU

Abbreviation for **HKEY_USERS**.

HKR

Relative to the registry key most pertinent to the section in which this **BitReg** directive appears, such as the per-device "hardware" subkey in the registry **..\Enum\***enumeratorID\ device-instance-id* branch, the corresponding driver-specific "software" subkey under the registry **..Class\***SetupClassGUID\device-instance-id* branch, and so forth.

subkey

This optional value, expressed either as a *%strkey%* token defined in a **Strings** section of the INF or as a registry path under the given *reg-root* (*key1\key2\key3...*), specifies the key containing the value entry to be modified.

value-entry-name

Specifies the name of an existing REG_BINARY-type value entry in the (existing) subkey to be modified. It can be expressed either as "*quoted string*" or as a *%strkey%* token that is defined in the INF's **Strings** section.

flag

This optional value specifies whether to clear or set the bits specified in the given *byte-mask*. Its default value is zero, meaning clear the bits. Specify one to set the bits.

byte-mask

This byte-sized mask, expressed in hexadecimal notation, specifies which bits to clear or set in the current value of the given *value-entry-name*.

byte-to-modify

This byte-sized value, expressed in decimal, specifies the zero-based index of the byte within the REG_BINARY-type value to be modified.

Comments

The value of an existing REG_BINARY-type value entry also can be modified by simply overwriting its current value within an add-registry section elsewhere in the INF file. For more information about add-registry sections, see the reference for the **AddReg** directive.

Using a **BitReg** directive requires the definition of another INF file section. However, the value of an existing REG_BINARY-type value entry can be modified bit-by-bit in such a section, thereby preserving the values of all remaining bits.

Examples

These examples show a bit-registry section for a fictional application.

```
[AppX_BitReg]
; set first bit of byte 0 in ProgramData value entry
HKLM,Software\AppX,ProgramData,1,0x01,0
; preceding would change value 30,00,10 to 31,00,10

; clear high bit of byte 2 in ProgramData value entry
HKLM,Software\AppX,ProgramData,,0x80,2
; preceding would change value 30,00,f0 to 30,00,70

; set second and third bits of byte 1 in ProgramData value entry
HKLM,Software\AppX,ProgramData,1,0x06,1
; preceding would change value 30,00,f0 to 30,06,f0
```

See Also

AddInterface, **AddReg**, **AddService**, **ClassInstall32**, ***DDInstall***, ***DDInstall*.CoInstallers**, ***DDInstall*.HW**, **InterfaceInstall32**

INF LogConfig Directive

[*DDInstall*] |
[***DDInstall*.LogConfigOverride**] |
[***DDInstall*.nt.LogConfigOverride**]

LogConfig=*log-config-section*[,*log-config-section*]...

A **LogConfig** directive typically references one or more INF-writer-defined sections in each of which a set of logical hardware configuration resources is specified. That is, each such section specifies the interrupt request lines, memory ranges, I/O ports, and DMA channels that can be used by the device (or set of compatible device models) to be installed. In effect, each such INF-writer-defined *log-config-section* presents an alternative set of bus-relative hardware resources that can be used by the given peripheral device(s).

INF files that install devices on nonPnP ISA, EISA, and MCA buses use this directive. INF files for PnP peripherals do not since the logical configurations of such devices is provided to their respective function drivers by the PnP Manager.

Comments

A **LogConfig** directive can be specified under any per-manufacturer, per-models *DDInstall* section, as shown here. Appending the **.LogConfigOverride** extension to a defined *DD-Install* section name forces the PnP Manager to accept the hardware configuration requirements supplied in the given *log-config-section*(s) of the INF for the given device(s), rather than using the resource requirements reported by the bus.

Otherwise, the PnP Manager assigns a set of logical hardware resources to each such device (or set of device-compatible models) covered by a *DDInstall* section. That is, the

PnP Manager queries the system bus drivers, receives their reports of per-device I/O bus configuration resources in use, and assigns per-device sets of logical hardware resources to achieve the best system-wide balance in the usage of all such resources.

The system-defined and case-insensitive extensions can be inserted into such a *DDInstall* (or *DDInstall*.**LogConfigOverride** or *DDInstall*.**nt.LogConfigOverride**) section containing a **LogConfig** directive in cross-OS and/or cross-platform INF files to specify platform-specific or OS-specific logical configurations. For more information about how to use the system-defined **.nt** extension in cross-platform and/or dual-OS INF files, see also *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide.*

In practice, an INF for any nonPnP device that supports several alternate logical configurations typically defines some number of log-config sections, each specifying a discrete set of logical configuration resources. Such an INF uses the **ConfigPriority** entry in each such log-config section to rank each possible logical configuration according to its effects on device/driver performance, ease of initialization, and so forth.

A given *log-config-section* name must be unique to the INF file, but it can be referenced by **LogConfig** directives in other INF *DDInstall.Xxx* sections for the same device(s). Each INF-writer-created section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files.*

Each named section referenced by a **LogConfig** directive has the following form:

[*log-config-section*]

ConfigPriority=*Priority_Value*[,*Config_Type*]
[DMAConfig=[*DMAattrs*:]*DMANum*[,*DMANum*]...]
[IOConfig=*io-range*[,*io-range*]...]
[**MemConfig=***mem-range*[,*mem-range*]...]
[IRQConfig=[*IRQattrs*:]*IRQNum*[,*IRQNum*]...]
[PcCardConfig=*ConfigIndex*[:[*MemoryCardBase1*][:*MemoryCardBase2*]][(*attrs*)]]
[**MfCardConfig=***ConfigRegBase*:*ConfigOptions*[:*IoResourceIndex*][(*attrs*)]...]
...

Only one **ConfigPriority** entry can be used in each log-config section. There can be more than one of each of the other entries, depending on the hardware resource requirements of the device.

One or more **MfCardConfig=** entries can appear only in a *log-config-section* that is refernced by a **LogConfig** directive in the *DDInstall*.**LogConfigOverride** section of an INF for a multifunction device. For more information about INFs for multifunction devices, see also the *Plug and Play, Power Management, and Setup Design Guide.*

LogConfig-Referenced Section Entries and Values

ConfigPriority=Priority_Value[,Config_Type]

Specifies the priority value for this logical configuration, as one of the following:

DESIRED

Soft configured, most optimal.

NORMAL

Soft configured, less optimal than DESIRED. This is the typical setting.

NORMAL should be specified if the log-config section was defined in a *DDInstall*.**LogConfigOverride** section, and no *Config_Type* value can be specified.

SUBOPTIMAL

Soft configured, less optimal than NORMAL.

HARDRECONFIG

Requires a jumper change to reconfigure.

HARDWIRED

Cannot be changed.

RESTART

Requires Windows 2000 restart to take effect.

REBOOT

For Windows 2000, this is the same as RESTART.

POWEROFF

Requires power cycle to take effect.

DISABLED

Hardware/device is disabled.

Config_Type is obsolete in Windows 2000 INF files.

DMAConfig=[DMAattrs:]DMANum[,DMANum]...]

DMAattrs is optional if the device is connected on a bus that has only 8-bit DMA channels and the device uses standard system DMA. Otherwise, it can be the letter **D** for 32-bit DMA, **W** for 16-bit DMA, or **N** for 8-bit DMA, with **M** if the device uses busmaster DMA and with one of the following (mutually exclusive) letters, indicating the type of DMA channel: **A**, **B**, or **F**. If **A**, **B**, or **F** is not specified, a standard DMA channel is assumed.

DMANum specifies one or more bus-relative DMA channel(s) as decimal number(s), each separated from the next by a comma (**,**).

IOConfig=io-range[,io-range]...

Specifies one or more I/O port range(s) for the device, in either of the following forms:

start-end[([*decode-mask*][:*alias-offset*][:*attr*])]
(Type 1 I/O range) where:

start specifies the starting address of the I/O port range as a 64-bit hexadecimal address.

end specifies the ending address of the I/O port range, also as a 64-bit hexadecimal address.

decode-mask defines the alias type and can be any of the following:

Mask Value	Meaning	IOR_Alias =
3ff	10-bit decode	0x04
fff	12-bit decode	0x10
ffff	16-bit decode	0x00
0	positive decode	0xFF

alias-offset is ignored.

attr specifies the letter **M** if the given range is in system memory. If omitted, the given range is in I/O port space.

size@min-max[%align-mask][([decode-mask][:alias-offset][:attr])]
(Type 2 I/O range) where:

size specifies the number of bytes required for the I/O port range as a 32-bit hexadecimal value.

min specifies the lowest possible starting address of the I/O port range as a 64-bit hexadecimal address.

max specifies the highest possible ending address of the I/O port range as a 64-bit hexadecimal address.

align-mask optionally specifies a 64-bit mask that is used in an AND operation to align the start of the I/O port range on an integral (usually 32-bit or 64-bit) address boundary.

decode-mask defines the alias type and can be any of the following:

Mask Value	Meaning	IOR_Alias =
3ff	10-bit decode	0x04
fff	12-bit decode	0x10
ffff	16-bit decode	0x00
0	positive decode	0xFF

alias-offset is ignored.

attr specifies the letter **M** if the given range is in system memory. If omitted, the given range is in I/O port space.

MemConfig=mem-range[,mem-range]...

Specifies one or more memory range(s) for the device in either of the following forms:

start-end[(attr)] | size@min-max[%align-mask][(attr)]
where:

start specifies the starting (bus-relative) physical address of the device memory range as a 64-bit hexadecimal value.

end specifies the ending physical address of the memory range, also as a 64-bit hexadecimal value.

attr specifies the attributes of the memory range as one or more of the following letters: **R** (read-only), **W** (write-only), **RW** (read/write), **C** (combined write allowed), **H** (cacheable), **F** (prefetchable), and **D** (card decode addressing is 32-bit, instead of 24-bit). If both **R** and **W** are specified or if neither is specified, read/write is assumed.

size specifies the number of bytes required in the memory range as a 32-bit hexadecimal value.

min specifies the lowest possible starting address of the device memory range as a 64-bit hexadecimal value.

max specifies the highest possible ending address of the memory range as a 64-bit hexadecimal value.

align-mask optionally specifies a 64-bit mask that is used in an AND operation to align the start of the device memory range on an integral (usually 64-bit) address boundary. If *align-mask* is omitted, the default memory alignment is on a 4K boundary (FFFFF000).

IRQConfig=[IRQattrs:]IRQNum[,IRQNum]...

IRQattrs is omitted if the device uses a bus-relative, edge-triggered IRQ. Otherwise, specify **L** to indicate a level-triggered IRQ and **LS** if the device can share the IRQ line(s) listed in this entry.

IRQNum specifies one or more bus-relative IRQs the device can use as decimal numbers, each separated from the next by a comma (,).

PcCardConfig=ConfigIndex[:[MemoryCardBase1][:MemoryCardBase2]][(attrs)]

Configures PCMCIA card registers and/or creates up to two permanent memory windows that map to the attribute space of the device. A driver can use the memory window(s) to access the attribute space from an ISR. Specify all numeric values in hexadecimal format.

The elements of a **PcCardConfig** entry are as follows:

ConfigIndex

Specifies the 8-bit PCMCIA configuration index for a device on a PCMCIA bus.

MemoryCardBase1

Optionally specifies a 32-bit base address for a first memory window.

MemoryCardBase2

Optionally specifies a 32-bit base address for a second memory window.

attrs

Optionally specifies one or more attributes for the device, separated by spaces. An invalid attribute specifier invalidates the whole **PcCardConfig** entry. Attributes can be specified in any order except for the positional attributes **A** and **C**.

The attributes include:

W	**16-bit I/O data path. The default is 8-bit if the INF specifies a LogConfig. If there is no LogConfig, the driver uses 16-bit I/O.**
S*n*	~IOCS16 source. If *n* is 0, ~IOCS16 is based on the value of the datasize bit. If *n* is 1, ~IOCS16 is based on the ~IOIS16 signal from the device. The default is **S**1.
Z*n*	I/O 8-bit, zero wait state. If *n* is 1, 8-bit I/O accesses occur with zero additional wait states. If *n* is 0, access will occur with additional wait states. This flag has no meaning for 16-bit I/O. The default is **Z**0.
XI*n*	I/O wait states. If n is 1, 16-bit system accesses occur with one additional wait state. The default is **X**1.
M	16-bit access to the memory windows. The default is 8-bit.
XM*n*	Memory wait states, where *n* can be 0, 1, 2 or 3. This value determines the number of additional wait states for 16-bit accesses to a memory window. The default is **XM**3.

The following two attributes relate positionally to memory window resources. The first **A** or **C** specified in the attribute string, reading from left to right, corresponds to the first memory resource in the device's resource list. The next **A** or **C** corresponds to the second memory resource. Subsequent attribute/common-memory specifiers are ignored.

A	**Memory range to be mapped as Attribute memory**
C	Memory range to be mapped as Common Memory (default)

For example, an *attrs* value of (W CA M XM1 XI0) translates to:

I/O 16-bit
1st memory window is common
2nd memory window is attribute
memory 16-bit
one wait state on memory windows
zero wait states on I/O windows

MfCardConfig=ConfigRegBase:ConfigOptions[:IoResourceIndex][(attrs)]...

Specifies the attribute-memory location of the set of configuration registers for one function of a multifunction device, as follows:

ConfigRegBase specifies the attribute offset of the configuration registers for this function of the multifunction device.

ConfigOptions specifies the 8-bit PCMCIA configuration option register.

IoResourceIndex specifies the index to the **IOConfig** entry for the bus driver to use in programming the configuration I/O base and limit registers. This index is zero-based, that is, zero designates the initial **IOConfig** entry in this log-config section.

attrs can be **A** if the PCMCIA bus driver should turn on audio enable in the configuration and status register(s).

Each **MfCardConfig** entry supplies information about a single function of the multifunction device. When a set of **LogConfig** directives each reference a discrete log-config section in the INF's *DDInstall*.**LogConfigOverride** section, each such log-config section must have its entries, including **MfCardConfig** entries, listed in the same order.

From a log-config section, the system installer builds binary logical configuration records and stores them in the registry.

An INF file can contain any number of per-device log-config sections. However, each such section must contain complete information for installing one device. In general, the INF should specify the entries in each of its log-config sections in the same order. The INF should specify each set of entries in the order best suited to how the driver initializes its device.

If more than one log-config section is present for a given device, only one of these INF sections will be used during installation. Such an INF file partially controls which such section is used with the **ConfigPriority** value it supplies in each such log-config section. That is, the system installer(s) attempt to honor any ranked log-config priorities in a given INF file, but might choose a lower ranked logical configuration if a conflict with an already installed device is found.

During installation, one and only one resource from each entry in a given log-config section is selected and assigned to a particular device. If a particular device needs more than one resource of the same type, a set of entries of that type must be used in its log-config section(s). For example, to ensure two I/O port ranges for a particular device, two **IOConfig=** entries must be specified in the log-config section for that device. On the other hand, if a device requires no IRQ, its INF can simply omit the **IRQConfig** entry from the log-config section(s).

Examples

This example shows some valid **PcCardConfig** entries for a PCMCIA device.

```
PcCardConfig=0:E0000:F0000(W)
PcCardConfig=0:E0000(M)
PcCardConfig=0::(W)
PcCardConfig=0(W)
```

This example shows a Type 1 I/O range specification in an **IOConfig** entry. It specifies an I/O port region, eight bytes in size, which can start at 1F8, 2F8, or 3F8.

```
IOConfig=1F8-1FF, 2F8-2FF, 3F8-3FF
```

By contrast, this example shows a Type 2 I/O range specification in an **IOConfig** entry. It specifies an I/O port region, eight bytes in size, which can start at 300, 308, 310, 318, 320, or 328.

```
IOConfig=8@300-32F%FF8
```

This example shows a set 258f **IOConfig=** entries for a four-port device, each specifying an I/O port range that is offset by 0x400 bytes from the next.

```
IoConfig=0x200-0x21f
IoConfig=0x600-0x61f
IoConfig=0xA00-0xA1f
IoConfig=0xE00-0xE1f
```

The next two examples show typical **MemConfig** entries.

This example specifies a memory region of 32K bytes that can start at either C0000 or D0000.

```
MemConfig=C0000-C7FFF, D0000-D7FFF
```

This example specifies a memory region of 32k bytes starting on 64K boundaries.

```
MemConfig=8000@C0000-D7FFF%F0000
```

This example shows how the system HDC class INF file sets up a number of log-config sections for generic ESDI hard disk controllers and uses a *DDInstall*.**LogConfigOverride** section for a particular IDE controller.

```
[MS_HDC] ; per-manufacturer Models section
%FujitsuIdePccard.DeviceDesc% =
        atapi_fujitsu_Inst, PCMCIA\FUJITSU-IDE-PC_CARD-DDF2
%*PNP0600.DeviceDesc% = atapi_Inst, *PNP0600 ; generic ESDI HDCs
%PCI\CC_0101.DeviceDesc% = pciide_Inst,,PCI\CC_0101

; ... other manufacturers' Models sections omitted

[atapi_Inst]
CopyFiles = @atapi.sys
LogConfig = esdilc1, esdilc2, esdilc3, esdilc4

; ... [atapi_Inst.Services] + service/EventLog-install omitted here

[esdilc1]
ConfigPriority=HARDWIRED
IOConfig=1f0-1f7(3ff::)
IoConfig=3f6-3f6(3ff::)
IRQConfig=14

[esdilc2]
ConfigPriority=HARDWIRED
IOConfig=170-177(3ff::)
IoConfig=376-376(3ff::)
IRQConfig=15

[esdilc3]
ConfigPriority=HARDWIRED
IOConfig=1e8-1ef(3ff::)
IoConfig=3ee-3ee(3ff::)
IRQConfig=11

[esdilc4]
; ...

[atapi_fujitsu_Inst.LogConfigOverride]
LogConfig = fujitsu.LogConfig0

[fujitsu.LogConfig0]
ConfigPriority=NORMAL
IOConfig=10@100-400%fff0
IRQConfig=14,15,5,7,9,11,12,3
PcCardConfig=1:0:0(W)
```

For some examples of how **MfCardConfig** entries are used, see also the *Plug and Play, Power Management, and Setup Design Guide.*

See Also

DDInstall, *DDInstall*.**FactDef**

INF ProfileItems Directive

[*DDInstall*]

ProfileItems=*profile-items-section*[,*profile-items-section*]...
...

A **ProfileItems** directive is used in a *DDInstall* section to list one or more *profile-items-sections* that contain items to be added to, or removed from, the Start menu.

This directive is only supported on Windows 2000.

Comments

A given *profile-items-section* name must be unique to the INF file.

Each INF-writer-defined section name must be unique within the INF and must follow the same general rules for defining section names described in *General Syntax Rules for INF Files*.

Each named section referenced by a **ProfileItems** directive has the following form:

[*profile-items-section*]

Name=*link-name*[,*name-attributes*]
CmdLine=*dirid*,[*subdir*],*filename*
[**SubDir=***path*]
[**WorkingDir=***wd-dirid*,*wd-subdir*]
[**IconPath=***icon-dirid*,[*icon-subdir*],*icon-filename*]
[**IconIndex=***index-value*]
[**HotKey=***hotkey-value*]
[**Infotip=***info-tip*]

Each *profile-items-section* contains detailed information for creating or removing one Start menu item. To manipulate more than one menu item from an INF, create more than one *profile-items-section* and list the sections in the **ProfileItems** directive.

Any of the string parameters specified in the *profile-items-section* entries can be specified using a *%strkey%* token, as described at the beginning of this chapter (*General Syntax Rules for INF Files*).

ProfileItems-Referenced Section Entries and Values

Name=link-name[,name-attributes]

The *link-name* specifies the name of the link for the menu item, without the *.lnk* extension.

The optional *name-attributes* value specifies one or more flags that affect the operation on the menu item. This value is expressed as an ORed bitmask of system-defined flag values. Possible flags include the following:

0x00000001 (= FLG_PROFITEM_CURRENTUSER)

Directs Setup to create or delete a Start menu item in the current user's profile. If this flag is not specified, Setup processes the item for all users.

0x00000002 (= FLG_PROFITEM_DELETE)

Directs Setup to delete the menu item. If this flag is not specified, the item is created.

0x00000004 (= FLG_PROFITEM_GROUP)

Directs Setup to create or delete a Start menu group. If this flag is not specified, Setup creates or deletes a menu item, not a menu group.

If no flag is specified, Setup creates a menu item for all users.

CmdLine=dirid,[subdir],filename

The *dirid* specifies a value that identifies the directory in which the command program resides. For example, a *dirid* of 11 indicates the system directory. The possible *dirid* values are listed in the description of the *dirid* value in the **DestinationDirs** section.

If a *subdir* string is present, the command program is in a subdirectory of the directory referenced by *dirid*. The *subdir* specifies the subdirectory. If no *subdir* is specified, the program resides in the directory referenced by *dirid*.

The *filename* specifies the name of the program associated with the menu item.

SubDir=path

This optional entry specifies a subdirectory (submenu) under Start\Programs in which the menu item resides. If this entry is omitted, the path defaults to Start\Programs.

For example, if the *profile-items-section* has the entry "**Subdir**=Accessories\Games", then the menu item is being created or deleted in the Start\Programs\Accessories\Games submenu.

WorkingDir=wd-dirid[,wd-subdir]

This optional entry specifies a working directory for the command program. If this entry is omitted, the working directory defaults to the directory in which the command program resides.

The *wd-dirid* value identifies the working directory. The possible *dirid* values are listed in the description of the *dirid* value in the **DestinationDirs** section.

The *wd-subdir* string, if present, specifies a subdirectory of *wd-dirid* to be the working directory. Use this parameter to specify a directory that doesn't have a system-defined *dirid.* If this parameter is omitted, the *wd-dirid* value alone specifies the working directory.

IconPath=icon-dirid,[icon-subdir],icon-filename

This optional entry specifies the location of a file that contains an icon for the menu item.

The *icon-dirid* string identifies the directory for the DLL that contains the icon. The possible *icon-dirid* values are listed in the description of the *dirid* value in the **DestinationDirs** section.

The *icon-subdir* value, if present, indicates that the DLL is in a subdirectory of *icon-dirid.* The *icon-subdir* value specifies the subdirectory.

The *icon-filename* value specifies the DLL that contains the icon.

If this entry is omitted, Setup looks for an icon in the file specified in the **CmdLine** entry.

IconIndex=index-value

This optional entry specifies which icon in a DLL to use for the menu item. For information on indexing the icons in a DLL, see the Platform SDK documentation.

If an **IconPath** entry is specified, the *index-value* indexes into that DLL. Otherwise, this value indexes into the file specified in the **CmdLine** entry.

HotKey=hotkey-value

This optional entry specifies a keyboard accelerator for the menu item. See the Platform SDK documentation for more information on hot keys.

Infotip=info-tip

This optional entry specifies an informational tip for the menu item. This value can be expressed as a string or as a *%strkey%* token that is defined in a **Strings** section of the INF file.

Example

The following INF excerpt illustrates how to use *profile-items-sections*.

```
:
[Freecell]
OptionDesc              = %Freecell_DESC%
Tip                     = %Freecell_TIP%
IconIndex               = 62 ;Windows mini-icon for dialogs
Parent                  = Games
CopyFiles               = FreecellCopyFilesSys, FreecellCopyFilesHelp
```

```
ProfileItems            = FreecellUninstallItems,FreecellInstallItems
Uninstall               = FreecellUninstall
Modes                   = 0,1,2,3

[FreecellUninstall]
DelFiles                = FreecellCopyFilesSys, FreecellCopyFilesHelp
ProfileItems            = FreecellUninstallItems, FreecellUninstallItemsCommon

[FreecellInstallItems]
Name       = %Freecell_DESC%
CmdLine    = 11,,freecell.exe
WorkingDir = 11
Subdir     = %Games_GROUP%
InfoTip    = %Freecell_Infotip%

[FreecellUninstallItems]
Name       = %Freecell_DESC%,0x00000003
Subdir     = %Games_GROUP%

[FreecellUninstallItemsCommon]
Name       = %Freecell_DESC%,0x00000002
Subdir     = %Games_GROUP%
:
:
[Strings]
KEY_OPTIONAL            = "SOFTWARE\Microsoft\Windows\CurrentVersion\Setup\Option
alComponents"

Games_DESC              = "Games"
Games_TIP               = "Includes Freecell, Minesweeper, Pinball, and Solitaire
 games.
Games_GROUP             = "Accessories\Games"

Freecell_DESC           = "Freecell"
Freecell_TIP            = "Logic puzzle in the form of a card game"

Minesweeper_DESC        = "Minesweeper"
Minesweeper_TIP         = "Strategy game"

Solitaire_DESC          = "Solitaire"
Solitaire_TIP           = "Card game"
```

See Also

DDInstall

INF UpdateInis Directive

[*DDInstall*] | [*DDInstall*.**CoInstallers**] | **[ClassInstall32]** | **[ClassInstall32.ntx86]**

UpdateInis=*update-ini-section*[, *update-ini-section*] ...

An **UpdateInis** directive references one or more named sections, specifying a given source INI file from which a particular section or line within such a section is to be read during installation and applied to an existing INI file of the same name on the target machine. Optionally, line-by-line modifications from and to such INI files can be specified in the update-ini section.

This directive is almost never specified in Windows 2000 device/driver INF files, due to the lack of necessity for INI files on their distribution media in prior Windows 2000 releases. However, the **UpdateInis** directive is valid in any of the sections shown in the formal syntax statement, as well as in INF-writer-defined sections referenced by an **AddInterface** directive or referenced in an **InterfaceInstall32** section.

Comments

A given *update-ini-section* name must be unique to the INF file.

Each INF-writer-created section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*. For more information about how to use the system-defined **.nt** and/or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide*.

Each named section referenced by an **UpdateInis** directive has the following form:

[*update-ini-section*]

ini-file,*ini-section*[,*old-ini-entry*][,*new-ini-entry*][,*flags*]
...

An *update-ini-section* can have any INF-writer-determined number of entries, each on a separate line.

UpdateInis-Referenced Section Entries

ini-file

Specifies the name of an INI file supplied on the source media and, implicitly, that of the INI file to be updated on the target machine. This value can be expressed as a *filename* or as a *%strkey%* token that is defined in a **Strings** section of the INF file.

ini-section

Specifies the name of the section within the given INI file. If the next two values are specified, this section contains an entry to be changed. If an *old-ini-entry* is omitted but a *new-ini-entry* is provided, the new entry is to be added as this section is read.

old-ini-entry

This optional value specifies the name of an entry in the given *ini-section*, usually expressed in the form "*key*=*value*". Either or both of *key* and *value* can be expressed as *%strkey%* tokens defined in a **Strings** section of the INF file. The asterisk (*) can be specified as a wild-card for either the *key* or the *value*.

new-ini-entry

This optional value specifies either a change to a given *old-ini-entry* or the addition of a new entry. This value can be expressed in the same manner as *old-ini-entry*.

flags

This optional value controls the interpretation of the given *old-ini-entry* and/or *new-ini-entry*. The *flags* can be one of the following numerical values:

Value	Meaning
0	This is the default value for the *flags* if it is omitted. If the given *old-ini-entry key* is present in the INI files, replace that *key*=*value* with the given *new-ini-entry*. Only the *keys* in the INI files must match; the corresponding *value* of each such *key* is ignored. To add a *new-ini-entry* to the destination INI file unconditionally, omit the *old-ini-entry* value from the entry in the update-ini section of the INF. To delete an *old-ini-entry* from the destination INI file unconditionally, omit the *new-ini-entry* value.
1	If the given *old-ini-entry* (*key*=*value*) exists in the INI files, replace it in the destination INI file with the given *new-ini-entry*. Both the *key* and *value* of the given *old-ini-entry* must match those in the INI files for such a replacement to be made, not just their *keys* as for the preceding *flags* value.
2	If the *key* specified for *old-ini-entry* cannot be found in the destination INI file, do nothing. Otherwise, the changes made depend on matches found in the INI files for the given *keys* of *old-ini-entry* and *new-ini-entry*, as follows: If the *key* of the *old-ini-entry* exists in the INI files but so does the *key* of the *new-ini-entry*, replace the *old-ini-entry* with the *new-ini-entry* in the destination INI file and, then, remove the superfluous *new-ini-entry* from that INI file. If the *key* of the *old-ini-entry* exists in the INI files but the *key* of the *new-ini-entry* does not, replace the *old-ini-entry key* with that of the *new-ini-entry* in the destination INI file but leave the *value* of the *old-ini-entry* unchanged.

Continued

Value	Meaning
3	If the *key* and *value* specified for *old-ini-entry* cannot be found in the INI files, do nothing. Otherwise, the changes made depend on matches found in the INI files for the given *key*s and *value*s of *old-ini-entry* and *new-ini-entry*, as follows: If the *key=value* of the *old-ini-entry* exists in the INI files but so does the *key=value* of the *new-ini-entry*, replace the *old-ini-entry* with the *new-ini-entry* in the destination INI file and, then, remove the superfluous *new-ini-entry* from that INI file. If the *key=value* of the *old-ini-entry* exists in the INI files but the *new-ini-entry* does not, replace the *old-ini-entry* with the *new-ini-entry* in the destination INI file but leave the *value* of the *old-ini-entry* unchanged.

The INF provides the full path to the given *ini-file* on the distribution media in one of the following ways:

- In IHV/OEM-supplied INFs, by using the **SourceDisksNames** and **SourceDisksFiles** sections of this INF to explicitly specify the full path to each named source file that is not in the root directory (or directories) on the distribution media
- In system-supplied INFs, by supplying one or more additional INF files, identified in the **LayoutFile** entry in the **Version** section of the INF file

Any *filename* specified within an *old-ini-entry* or *new-ini-entry* should designate the destination directory containing that file. Such a destination directory path to a *filename* in an *update-ini-section* entry must be specified as the value of a DIRID_*XXX*, enclosed in *%* (per cent) characters and followed by a \ (back slash). That is, such a DIRID_*XXX* specification in an update-ini section entry has the form *%dirid%* where *dirid* is one of the predefined directory identifiers or is a user-defined value created by a call to the **SetupSetDirectoryId** function. The backslash separates the *%dirid%* from the given *filename*, as in any full path specification to a file. For example, *%11%\card.ini* can be used to reference *card.ini* in the *system32* directory. The installer replaces such a specification with the full destination path to such a file during installation.

For a summary of the predefined *dirid* values, see the reference for the **DestinationDirs** section.

Example

The following example shows, in part, how the **UpdateInis** directive is used to install the system-supplied FAX services. This method works on both Windows 98 and Windows 2000. If your INF only supports Windows 2000, consider using the **ProfileItems** directive instead. (See the *INF ProfileItems Directive* section for more information.)

```
[FAX]
; ... some other directives omitted for brevity
UpdateInis = FAXInis, OLDFaxRemove.ini, WFWFaxCleanUp.ini ; ...
; ...
[FAXInis]
; Create link to SendTo folder, but first create folder
setup.ini, progman.groups,, "SendTo=""..\..\%SendTo_Desc%"""
; Create link to EXE
setup.ini,SendTo,,
       """%SendToFax_DESC%"",%11%\awsnto32.exe,,,,,%Sendfax%"
;
; Program Item for the FAX group
; first, create Accessories/Fax folder (if it doesn't exist)
;
setup.ini,progman.groups,, "groupFAX=%FAXApps_DESC%"
setup.ini,groupFax,,\       """%FAXCOVER_Link_Desc%"",FaxCover.exe,,,,,%FaxCover
%"
; ...
setup.ini,groupFAX,,\      """%SendNewFax%"",%11%\awsnto32.exe,,,,,%Newfax%"
; ... more "update" entries omitted here
```

See Also

AddInterface, **ClassInstall32**, ***DDInstall***, ***DDInstall*.CoInstallers**, **DestinationDirs**, **Ini2Reg**, **InterfaceInstall32**, **ProfileItems**, **SourceDisksFiles**, **SourceDisksNames**, **Strings**, **UpdateIniFields**, **Version**

INF UpdateIniFields Directive

[*DDInstall*] | [*DDInstall*.**CoInstallers]** | **[ClassInstall32]** | **[ClassInstall32.ntx86]**

UpdateIniFields=*update-inifields-section*[**,** *update-inifields-section*] ...

An **UpdateIniFields** directive references one or more named sections in which fine-grained modifications within the lines of an INI file can be specified.

This directive is almost never specified in Windows 2000 device/driver INF files, due to the lack of necessity for INI files on their distribution media in prior Windows 2000 releases. However, the **UpdateIniFields** directive is valid in any of the sections shown in the formal syntax statement, as well as in INF-writer-defined sections referenced by an **AddInterface** directive or referenced in an **InterfaceInstall32** section.

Comments

A given *update-inifields-section* name must be unique to the INF file. Each INF-writer-created section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files*. For more information about how to use the system-defined **.nt** and/or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide.*

Each named section referenced by an **UpdateIniFields** directive has the following form:

[*update-inifields-section*]

ini-file,*ini-section*,*profile-name*[,*old-field*][,*new-field*][,*flags*]
...

An *update-inifields-section* can have any INF-writer-determined number of entries, each on a separate line.

By contrast with a section referenced by the **UpdateInis** directive, a section referenced by **UpdateIniFields** replaces, adds, or deletes portions of a line in an existing INI file line rather than affecting the whole value of a particular line. At least one of the *old-field* and/or *new-field* value(s) must be specified in each section entry.

UpdateIniFields-Referenced Section Entries

ini-file

Specifies the name of an INI file supplied on the source media and, implicitly, that of a to-be-updated INI file on the target machine. This value can be expressed as a *filename* or as a *%strkey%* token that is defined in a **Strings** section of the INF file.

ini-section

Specifies the name of the section within the given INI files containing the line to be modified.

profile-name

Specifies the name of the line to be modified within the given INI section. At least one of the *old-field* and/or *new-field* must be specified to effect a modification of this line.

old-field

Specifies an existing field within the given line. If *new-field* is omitted from this section entry, this field will be deleted from the given line. Otherwise, the given *new-field* value should replace this field.

new-field

Specifies a replacement for a given *old-field* or, if *old-field* is omitted, an addition to the given line.

flags

Specifies (in bit 0) how to interpret given *old-field* and/or *new-field* if either or both contain an asterisk (*), and/or (in bit 1) which separator character to use when appending a given *new-field* to the given line, as follows:

Bit zero = 0

Interpret any asterisk (*) in the given *old-field* and/or *new-field* literally, not as a wild-card character, when searching for a match in the given line of the INI file. This is the default value.

Bit zero = 1

Interpret any asterisk (*) in the given *old-field* and/or *new-field* as a wild-card character when searching for a match in the given line of the INI file.

Bit one = 0

Use a space character as a separator when adding the given *new-field* to the given line of the INI file. This is the default value.

Bit one = 1

Use a comma (,) as a separator when adding the given *new-field* to the given line of the INI file.

Any comments in a to-be-modified INI file line are removed because they might not be applicable after changes made according to this section. When looking for fields in the line in the INI files, spaces, tabs, and commas are interpreted as field delimiters. However, a space character is used as the default separator when a new field is appended to a line.

The INF provides the full path to the given *ini-file* on the distribution media in one of the following ways:

- In IHV/OEM-supplied INFs, by using the **SourceDisksNames** and **SourceDisksFiles** sections of this INF to explicitly specify the full path to each named source file that is not in the root directory (or directories) on the distribution media
- In system-supplied INFs, by supplying one or more additional INF files, identified in the **LayoutFile** entry in the **Version** section of the INF file

See Also

AddInterface, **ClassInstall32**, ***DDInstall***, ***DDInstall*.CoInstallers**, **Ini2Reg**, **Interface-Install32**, **SourceDisksFiles**, **SourceDisksNames**, **Strings**, **UpdateInis**, **Version**

INF Ini2Reg Directive

[*DDInstall*] | [*DDInstall*.**CoInstallers**] | **[ClassInstall32]** | **[ClassInstall32.ntx86]**

Ini2Reg=*ini-to-registry-section*[**,** *ini-to-registry-section*] ...

An **Ini2Reg** directive references one or more named sections in which lines or sections from a supplied INI file are moved into the registry, thereby creating or replacing one or more value entries under a specified key.

This directive is almost never specified in Windows 2000 device/driver INF files, due to the lack of necessity for INI files on their distribution media in prior Windows NT/Windows 2000 releases. However, the **Ini2Reg** directive is valid in any of the sections shown in the formal syntax statement, as well as in INF-writer-defined sections referenced by an **AddInterface** directive or referenced in an **InterfaceInstall32** section.

Comments

A given *ini-to-registry-section* name must be unique to the INF file. Each INF-writer-created section name must be unique within the INF and must follow the same general rules for defining section names already described in *General Syntax Rules for INF Files.* For more information about how to use the system-defined **.nt** and/or **.ntx86** extensions in cross-platform and/or dual-OS INF files, see also *Creating an INF File* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide.*

Each named section referenced by an **Ini2Reg** directive has the following form:

[*ini-to-registry-section*]

*ini-file***,***ini-section***,**[*ini-key*]**,***reg-root***,***subkey*[**,***flags*]
...

An *ini-to-registry-section* can have any INF-writer-determined number of entries, each on a separate line.

Ini2Reg-Referenced Section Entries

ini-file

Specifies the name of an INI file supplied on the source media. This value can be expressed as a *filename* or as a *%strkey%* token that is defined in a **Strings** section of the INF file.

ini-section

Specifies the name of the section within the given INI file containing the registry information to be copied.

ini-key

Specifies the name of the key in the INI file to copy to the registry. If this value is omitted, the whole *ini-section* is to be transferred to the specified registry *subkey*.

reg-root

Identifies the root of the registry tree for other values supplied in this entry. For specifics, see the reference for the **AddReg** directive.

subkey

Identifies the subkey to receive the value, expressed either as a *%strkey%* token defined in a **Strings** section of the INF or as an explicit registry path (*key1\key2\key3...*) from the given *reg-root*.

flags

Specifies (in bit 0) how to handle the INI file after transferring the given information to the registry and/or (in bit 1) whether to overwrite existing registry information, as follows:

Bit zero = 0

Do not remove the given information from the INI file after copying it into the registry. This is the default.

Bit zero = 1

Delete the given information from the INI file after moving it into the registry.

Bit one = 0

If the specified *subkey* already exists in the registry, do not transfer the INI-supplied information into this subkey. Otherwise, create the given *subkey* in the registry with this INI-supplied information as its value entry. This is the default.

Bit one = 1

If the specified subkey already exists in the registry, replace its value entry with the INI-supplied information.

The INF provides the full path to the given *ini-file* on the distribution media in one of the following ways:

- In IHV/OEM-supplied INFs, by using the **SourceDisksNames** and, possibly, **Source-DisksFiles** sections of this INF to explicitly specify the full path to each named source file that is not in the root directory (or directories) on the distribution media
- In system-supplied INFs, by supplying one or more additional INF files, identified in the **LayoutFile** entry in the **Version** section of the INF file

See Also

AddInterface, AddReg, ClassInstall32, *DDInstall*, *DDInstall*.CoInstallers, Interface-Install32, SourceDisksFiles, SourceDisksNames, Strings, UpdateIniFields, UpdateInis, Version

C H A P T E R 2

Setup Functions

This chapter summarizes the Setup functions. Installation software can use these functions to:

- Read and process INF files
- Determine the amount of free space required on the installation's target system
- Move files from installation source media to media on the installation's target system, while requesting user intervention as needed
- Create a log of files moved during an installation

Installation software typically uses these functions in conjunction with *device installation functions* and *PnP Configuration Manager functions*.

The Setup functions listed in this chapter are described in detail in the Platform SDK documentation. Their purpose is to enable installation software.

This chapter contains the following topics:

- *INF File Processing Functions*
- *Disk Prompting and Error Handling Functions*
- *File Queuing Functions*
- *Default Queue Callback Routine Functions*
- *Cabinet File Function*
- *Disk-Space List Functions*
- *MRU Source List Functions*
- *File Log Functions*

INF File Processing Functions

The INF file processing functions provide setup and installation functionality that includes:

- Opening and closing an INF file
- Retrieving information about an INF file
- Retrieving information about source files and target directories for copy operations
- Performing the installation actions specified in an INF file section

The following table lists the functions used for processing INF files. For detailed function descriptions, see the Platform SDK documentation.

Function	Description
SetupCloseInfFile	Frees resources and closes the INF handle.
SetupCopyOEMInf	Copies a file into *%windir%\Inf.*
SetupDecompressOrCopyFile	Copies a file and, if necessary, decompresses it.
SetupFindFirstLine	Finds a pointer to the first line in a section of an INF file or, if a key is specified, the first line that matches the key.
SetupFindNextLine	Returns a pointer to the next line in an INF file section.
SetupFindNextMatchLine	Returns a pointer to the next line in an INF file section or, if a key is specified, the next line that matches the key.
SetupGetBinaryField	Retrieves binary data from a field in a specified line, in an INF file.
SetupGetFieldCount	Returns the number of fields in a line.
SetupGetFileCompressionInfo	Retrieves file compression information from an INF file.
SetupGetInfFileList	Returns a list of the INF files in a specified directory.
SetupGetInfInformation	Returns information about an INF file.
SetupGetIntField	Obtains the integer value of a specified field in a specified line, in an INF file.
SetupGetLineByIndex	Returns a pointer to the line associated with a specified index value in a specified section.
SetupGetLineCount	Returns the number of lines in the specified section.
SetupGetLineText	Retrieves the contents of a specified line from an INF file.
SetupGetMultiSzField	Returns multiple strings, starting at a specified field in a line.
SetupGetSourceFileLocation	Returns the location of a source file listed in an INF file.
SetupGetSourceFileSize	Returns the size of a specified file or a set of files listed in a specified section of an INF file.
SetupGetSourceInfo	Retrieves the path, tag file, or description for a source.

Function	Description
SetupGetStringField	Retrieves string data from a field in a specified line, in an INF file.
SetupGetTargetPath	Determines the target path for the files listed in a specified INF file section.
SetupInstallFile	Installs a specified file into a specific target directory.
SetupInstallFileEx	Installs a specified file into a specific target directory. The installation is postponed if an existing version of the file is in use.
SetupInstallFilesFromInfSection	Queues the files in a specified INF file section for copying. (Same as **SetupQueueCopySection**.)
SetupInstallFromInfSection	Performs the directives specified in an INF *DDInstall* section.
SetupInstallServicesFromInf-Section	Performs service installation and deletion operations as specified in an INF *DDInstall*.**Services** section.
SetupOpenAppendInfFile	Opens an INF file and appends it to an existing INF handle.
SetupOpenInfFile	Opens an INF file and returns a handle to it.
SetupOpenMasterInf	Opens the master INF file that contains file and layout information for files shipped with Microsoft® Windows NT®/Windows® 2000.
SetupQueryInfFileInformation	Returns the name of one of the constituent INF files of a specified INF file.
SetupQueryInfVersion-Information	Returns the version number of one of the constituent INF files of a specified INF file.
SetupSetDirectoryId	Assigns a directory ID (DIRID) to a specified directory.

Disk Prompting and Error Handling Functions

You can use the Setup functions to prompt the user to insert new media, or to handle errors that arise when files are being copied, renamed, or deleted.

The following table lists functions that provide dialog boxes for requesting installation media and reporting errors. For detailed function descriptions, see the Platform SDK documentation.

Function	Description
SetupCopyError	Generates a dialog box that informs the user of a copy error.
SetupDeleteError	Generates a dialog box that informs the user of a delete error.
SetupPromptForDisk	Generates a dialog box that prompts the user for an installation medium or source file location.
SetupRenameError	Generates a dialog box that informs the user of a rename error.

File Queuing Functions

Using the Setup functions, you can queue files for various operations. File queues can be established for copying, renaming, and deleting files. Typically, an application queues all of the file operations necessary for an entire installation, then "commits" the queue so the operations are performed in a single batch.

The following table provides a summary of file queuing functions. For detailed function descriptions, see the Platform SDK documentation.

Function	Description
SetupCloseFileQueue	Destroys a file queue along with any uncommitted file operations.
SetupCommitFileQueue	Commits (performs) all queued operations.
SetupOpenFileQueue	Creates and returns a handle to a file queue.
SetupPromptReboot	Prompts the user to reboot his or her machine, if necessary.
SetupQueueCopy	Queues a specified file for copying.
SetupQueueCopySection	Queues the files in a specified INF file section for copying.
SetupQueueDefaultCopy	Queues a specified file for copying, using default source and destination settings contained in the INF file.
SetupQueueDelete	Queues a specified file for deletion.
SetupQueueDeleteSection	Queues the files in an INF file section for deletion.
SetupQueueRename	Queues a specified file for renaming.
SetupQueueRenameSection	Queues the files in an INF section for renaming.
SetupScanFileQueue	Scans a file queue and performs a specified operation on each queue entry.
SetupSetPlatformPath Override	Sets the value used for overriding the default platform-specific source path.

Default Queue Callback Routine Functions

If you associate a callback routine with a file queue, the callback routine will be called each time the system performs one of the queued file operations. Typically, you can use the default queue callback routine, **SetupDefaultQueueCallback**, to handle these notifications.

The following table lists functions associated with the default queue callback routine. For detailed function descriptions, and for more information about using callback routines with file queues, see the Platform SDK documentation.

Function	Description
SetupDefaultQueueCallback	Handles notifications sent by the system when queued file operations are performed.
SetupInitDefaultQueueCallback	Initializes context information needed by **Setup-DefaultQueueCallback**.
SetupInitDefaultQueueCallbackEx	Initializes context information needed by **Setup-DefaultQueueCallback**, and provides a separate window for displaying progress messages.
SetupTermDefaultQueueCallback	Notifies the system that the setup application will not commit any more file queue operations.

Cabinet File Function

A cabinet file is a single file, usually with a .cab extension, that contains several compressed files as a file library. Cabinet files are used to organize the installation files that will be copied to the user's system. A compressed file can be spread over several cabinet files.

The following function is used with cabinet files. For a detailed function description, see the Platform SDK documentation.

Function	Description
SetupIterateCabinet	Sends a notification to a callback function for each file stored in a cabinet file.

Disk-Space List Functions

Disk-space list functions are used to create and modify disk-space lists. These lists can be used to calculate the total disk space required to handle the files that will be copied or deleted during the installation procedure.

The following table lists the functions that can be used to manipulate disk-space lists. For detailed function descriptions, see the Platform SDK documentation.

Function	Description
SetupAddInstallSectionToDisk-SpaceList	Searches for **CopyFile** and **DelFile** directives in a *DDInstall* section of an INF file, then adds the file operations specified in those sections to a disk-space list.
SetupAddSectionToDisk-SpaceList	Adds to a disk-space list all the file copy or delete operations listed in a specified section of an INF file.
SetupAddToDiskSpaceList	Adds a single delete or copy operation to a disk-space list.

Continued

Function	Description
SetupCreateDiskSpaceList	Creates a disk-space list.
SetupDestroyDiskSpaceList	Destroys a disk-space list and releases the resources allocated to it.
SetupQueryDrivesInDiskSpace-List	Fills a caller-supplied buffer with a list of the drives referenced by the file operations listed in the disk-space list.
SetupQuerySpaceRequiredOn-Drive	Examines a disk-space list to determine the space required to perform all the file operations listed for a particular drive.
SetupRemoveFromDiskSpace-List	Removes a file copy or delete operation from a disk-space list.
SetupRemoveInstallSection-FromDiskSpaceList	Searches for **CopyFiles** and **DelFiles** directives in a *DDInstall* section of an INF file, and removes the file operations specified in those sections from a disk-space list.
SetupRemoveSectionFromDisk-SpaceList	Removes from a disk-space list the file copy or delete operations listed in a specified section of an INF file.

MRU Source List Functions

Most recently used (MRU) source lists are resident on the user's machine and contain information about source paths used in previous installations. This information can be used when prompting the user for a source path.

The setup application can access a user-specific source list and, if the application has administrator privilege, the system-wide source list. The setup application can also create a temporary source list that is discarded when the setup application exits. By calling **SetupSet-SourceList**, the setup application identifies which source list it will use during the installation.

The following table lists the functions that can be used to manipulate source lists. For detailed function descriptions, see the Platform SDK documentation.

Function	Description
SetupAddToSourceList	Adds an entry to a source list.
SetupCancelTemporarySourceList	Cancels use of a temporary list.
SetupFreeSourceList	Frees resources allocated by a previous call to **Setup-SetSourceList**.
SetupQuerySourceList	Queries the current list of installation sources.
SetupRemoveFromSourceList	Removes an entry from an installation source list.
SetupSetSourceList	Sets the installation source list to the system MRU list, the user MRU list, or a temporary list.

File Log Functions

You can use a log file to record information about the files copied to a system during an installation. The log file can be either the system log or your own installation log file.

The following table lists the functions that can be used to manipulate log files. For detailed function descriptions, see the Platform SDK documentation.

Function	Description
SetupInitializeFileLog	Initializes a log file for use.
SetupLogError	Writes an error message to a log file. (It should be used only during the installation of the operating system.)
SetupLogFile	Adds an entry to the log file.
SetupQueryFileLog	Retrieves information from a log file.
SetupRemoveFileLogEntry	Removes an entry from a log file.
SetupTerminateFileLog	Releases resources allocated to a log file.

C H A P T E R 3

Device Installation Functions

This section describes the Setup functions that support Windows® 2000 device installation. These functions provide high-level device installation support used by system installation and maintenance utilities. They provide a superset of the functionality provided by the Windows 98 Device Installer functions. For the device installation operations that do not have SetupDi*Xxx* APIs, call the appropriate *PnP Configuration Manager Functions* (CM_*Xxx* functions*)*.

This chapter lists the Setup device installation functions in alphabetical order. The following tables provide functional summaries of the functions:

- Update Driver Function
- Device Information Functions
- Driver Information Functions
- Driver Selection Functions
- Device Installation Handlers
- Device Installation Customization Functions
- Setup Class Functions
- Class Bitmap and Icon Functions
- Device Interface Functions
- Registry Functions
- Other Functions

Update Driver Function

Function	Description
UpdateDriverForPlugAndPlayDevices	Given an INF and a hardware ID, UpdateDriver-ForPlugAndPlayDevices installs updated drivers for devices that match the hardware ID.

SetupDi Device Information Functions

Function	Description
SetupDiCreateDeviceInfoList	Creates an empty device information set. This set can be associated with a class GUID.
SetupDiCreateDeviceInfoListEx	Creates an empty device information set. This set can be associated with a class GUID and can be for devices on a remote machine.
SetupDiCreateDeviceInfo	Creates a new device information element and adds it as a new member to the specified device information set.
SetupDiOpenDeviceInfo	Retrieves information about an existing device instance and adds it to the specified device information set.
SetupDiEnumDeviceInfo	Returns a context structure for a device information element of a device information set.
SetupDiGetDeviceInstanceId	Retrieves the device instance ID associated with a device information element.
SetupDiGetDeviceInfoListClass	Retrieves the class GUID associated with a device information set if it has an associated class.
SetupDiGetDeviceInfoListDetail	Retrieves information associated with a device information set including the class GUID, remote machine handle, and remote machine name.
SetupDiGetClassDevs	Returns a device information set that contains all devices of a specified class.
SetupDiGetClassDevsEx	Returns a device information set that contains all devices of a specified class on a local or remote machine.
SetupDiSetSelectedDevice	Sets the specified device information element to be the currently-selected member of a device information set. This function is typically used by an installation wizard.
SetupDiGetSelectedDevice	Retrieves the currently-selected device for the specified device information set.

Function	Description
SetupDiRegisterDeviceInfo	Registers a newly created device instance with the Plug and Play Manager.
SetupDiDeleteDeviceInfo	Deletes a member from the specified device information set. This function does not delete the actual device.
SetupDiDestroyDeviceInfoList	Destroys a device information set and frees all associated memory.

SetupDi Driver Information Functions

Function	Description
SetupDiBuildDriverInfoList	Builds a list of drivers associated with a specified device instance or with the device information set's global class driver list.
SetupDiEnumDriverInfo	Enumerates the members of a driver information list.
SetupDiGetDriverInfoDetail	Retrieves detailed information for a specified driver information element.
SetupDiSetSelectedDriver	Sets the specified member of a driver list as the currently selected-driver. It can also be used to reset the driver list so that there is no currently-selected driver.
SetupDiGetSelectedDriver	Retrieves the member of a driver list that has been selected as the driver to install.
SetupDiCancelDriverInfoSearch	Cancels a driver list search that is currently underway in a different thread.
SetupDiDestroyDriverInfoList	Destroys a driver information list.

SetupDi Driver Selection Functions

Function	Description
SetupDiAskForOEMDisk	Displays a dialog that asks the user for the path to an OEM installation disk.
SetupDiSelectOEMDrv	Selects a driver for a device using an OEM path supplied by the user.
SetupDiSelectDevice	Default handler for the DIF_SELECTDEVICE request.

SetupDi Device Installation Handlers

Function	Description
SetupDiCallClassInstaller	Calls any registered coinstallers and the appropriate class installer with the specified installation request.
SetupDiChangeState	The default handler for the DIF_PROPERTY-CHANGE request. It can be used to change the state of an installed device.
SetupDiRegisterCoDeviceInstallers	Registers the device-specific coinstallers listed in the INF file for the specified device. This function is the default handler for DIF_REGISTER_COINSTALLERS.
SetupDiInstallDevice	The default handler for the DIF_INSTALLDEVICE request.
SetupDiInstallDriverFiles	The default handler for the DIF_INSTALLDEVICE-FILES request.
SetupDiInstallDeviceInterfaces	The default handler for the DIF_INSTALL-INTERFACES request. It installs the interfaces listed in a *DDInstall*.**Interfaces** section of a device INF file.
SetupDiMoveDuplicateDevice	The default handler for the DIF_MOVEDEVICE request.
SetupDiRemoveDevice	The default handler for the DIF_REMOVEDEVICE request.
SetupDiUnremoveDevice	The default handler for the DIF_UNREMOVE request.
SetupDiRegisterDeviceInfo	The default handler for the DIF_REGISTERDEVICE request.
SetupDiSelectDevice	The default handler for the DIF_SELECTDEVICE request.
SetupDiSelectBestCompatDrv	The default handler for the DIF_SELECTBEST-COMPATDRV request.

SetupDi Device Installation Customization Functions

Function	Description
SetupDiGetClassInstallParams	Retrieves class install parameters for a device information set or a particular device information element.
SetupDiSetClassInstallParams	Sets or clears class install parameters for a device information set or a particular device information element.

Function	Description
SetupDiGetDeviceInstallParams	Retrieves device install parameters for a device information set or a particular device information element.
SetupDiSetDeviceInstallParams	Sets device install parameters for a device information set or a particular device information element.
SetupDiGetDriverInstallParams	Retrieves install parameters for the specified driver.
SetupDiSetDriverInstallParams	Sets the install parameters for the specified driver.

SetupDi Setup Class Functions

Function	Description
SetupDiBuildClassInfoList	Returns a list of setup class GUIDs that includes every class installed on the system.
SetupDiBuildClassInfoListEx	Returns a list of setup class GUIDs that includes every class installed on the local system or a remote system.
SetupDiGetClassDescription	Retrieves the class description associated with the specified setup class GUID.
SetupDiGetClassDescriptionEx	Retrieves the description of a setup class installed on a local or remote machine.
SetupDiGetINFClass	Retrieves the class of a specified device INF file.
SetupDiClassGuidsFromName	Retrieves the GUID(s) associated with the specified class name. This list is built based on what classes are currently installed on the system.
SetupDiClassGuidsFromNameEx	Retrieves the GUID(s) associated with the specified class name. This resulting list contains the classes currently installed on a local or remote machine.
SetupDiClassNameFromGuid	Retrieves the class name associated with the class GUID.
SetupDiClassNameFromGuidEx	Retrieves the class name associated with a class GUID. The class can be installed on a local or remote machine.
SetupDiInstallClass	Installs the **ClassInstall32** section of the specified INF file.
SetupDiInstallClassEx	Installs a class installer or an interface class.
SetupDiOpenClassRegKey	Opens the device setup class registry key or a specific class's subkey.

Continued

Function	Description
SetupDiOpenClassRegKeyEx	Opens the device setup class registry key, the device interface class registry key, or a specific class's subkey. This function opens the specified key on the local machine or on a remote machine.

SetupDi Class Bitmap and Icon Functions

Function	Description
SetupDiGetClassImageList	Builds an image list that contains bitmaps for every installed class and returns the list in a data structure.
SetupDiGetClassImageListEx	Builds an image list of bitmaps for every class installed on a local or remote machine.
SetupDiGetClassImageIndex	Retrieves the index within the class image list of a specified class.
SetupDiGetClassBitmapIndex	Retrieves the index of the mini-icon supplied for the specified class.
SetupDiDrawMiniIcon	Draws the specified mini-icon at the location requested.
SetupDiLoadClassIcon	Loads both the large and mini-icon for the specified class.
SetupDiDestroyClassImageList	Destroys a class image list.

SetupDi Device Interface Functions

Function	Description
SetupDiCreateDeviceInterface	Registers device functionality (a device interface) for a device.
SetupDiOpenDeviceInterface	Retrieves information about an existing device interface and adds it to the specified device information set.
SetupDiGetDeviceInterfaceAlias	Returns an alias of the specified device interface.
SetupDiGetClassDevs	Returns a device information set that contains all devices of a specified class.
SetupDiGetClassDevsEx	Returns a device information set that contains all devices of a specified class on a local or remote machine.

Function	Description
SetupDiEnumDeviceInterfaces	Returns a context structure for a device interface element of a device information set. Each call returns information about one device interface; the function can be called repeatedly to get information about several interfaces exposed by one or more devices.
SetupDiGetDeviceInterfaceDetail	Returns details about a particular device interface.
SetupDiCreateDeviceInterfaceRegKey	Creates a registry subkey for storing information about a device interface instance and returns a handle to the key.
SetupDiOpenDeviceInterfaceRegKey	Opens the registry subkey that is used by applications and drivers to store information specific to a device interface instance and returns a handle to the key.
SetupDiDeleteDeviceInterfaceRegKey	Deletes the registry subkey that was used by applications and drivers to store information specific to a device interface instance.
SetupDiInstallDeviceInterfaces	Is the default handler for the DIF_ INSTALLINTERFACES request. It installs the interfaces listed in a *DDInstall*.**Interfaces** section of a device INF file.
SetupDiRemoveDeviceInterface	Removes a registered device interface from the system.
SetupDiDeleteDeviceInterfaceData	Deletes a device interface from a device information set.
SetupDiInstallClassEx	Installs a class installer or an interface class.
SetupDiOpenClassRegKeyEx	Opens the device setup class registry key, the device interface class registry key, or a specific class's subkey. This function opens the specified key on the local machine or on a remote machine.

SetupDi Registry Functions

Function	Description
SetupDiCreateDevRegKey	Creates a registry storage key for device-specific configuration information and returns a handle to the key.
SetupDiOpenDevRegKey	Opens a registry storage key for device-specific configuration information and returns a handle to the key.

Continued

Function	Description
SetupDiDeleteDevRegKey	Deletes the specified user-accessible registry key(s) associated with a device information element.
SetupDiOpenClassRegKey	Opens the setup class registry key or a specific class's subkey.
SetupDiOpenClassRegKeyEx	Opens the device setup class registry key, the device interface class registry key, or a specific class's subkey. This function opens the specified key on the local machine or on a remote machine.
SetupDiCreateDeviceInterfaceRegKey	Creates a non-volatile registry subkey for storing information about a device interface instance and returns a handle to the key.
SetupDiOpenDeviceInterfaceRegKey	Opens the registry subkey that is used by applications and drivers to store information specific to a device interface instance and returns a handle to the key.
SetupDiDeleteDeviceInterfaceRegKey	Deletes the registry subkey that was used by applications and drivers to store information specific to a device interface instance.
SetupDiSetDeviceRegistryProperty	Sets the specified Plug and Play device property.
SetupDiGetDeviceRegistryProperty	Retrieves the specified Plug and Play device property.

Other SetupDi Functions

Function	Description
SetupDiGetActualSectionToInstall	Finds the appropriate *DDInstall* section to use when installing a device from a device INF file.
SetupDiGetHwProfileFriendlyName	Retrieves the friendly name associated with a hardware profile ID.
SetupDiGetHwProfileFriendlyNameEx	Retrieves the friendly name associated with a hardware profile ID on a local or remote machine.
SetupDiGetHwProfileList	Retrieves a list of all currently defined hardware profile IDs.
SetupDiGetHwProfileListEx	Retrieves a list of all currently defined hardware profile IDs on a local or remote machine.

SetupDiAskForOEMDisk

```
BOOLEAN
  SetupDiAskForOEMDisk(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
    );
```

SetupDiAskForOEMDisk displays a dialog that asks the user for the path to an OEM installation disk.

Parameters

DeviceInfoSet

Supplies a handle to the device information set for the local machine that contains the device being installed.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure for the device being installed. If this parameter is not specified, the driver being installed is associated with the global class list of the device information set.

Return Value

The function returns TRUE if it is successful and the **DriverPath** field of the SP_DEV-INSTALLPARAMS structure is updated to reflect the new path. If the user cancels the dialog, the function returns FALSE and a call to **GetLastError** returns ERROR_ CANCELLED.

Comments

This function allows the user to browse local and network drives for OEM installation files. The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiSelectOEMDrv

SetupDiBuildClassInfoList

```
BOOLEAN
  SetupDiBuildClassInfoList(
    IN DWORD  Flags,
    OUT LPGUID  ClassGuidList,
    IN DWORD  ClassGuidListSize,
    OUT PDWORD  RequiredSize
    );
```

SetupDiBuildClassInfoList returns a list of setup class GUIDs that includes every class installed on the system.

Parameters

Flags

Flags used to control exclusion of classes from the list. If no flags are specified, all setup classes are included in the list. Can be a combination of the following values:

DIBCI_NOINSTALLCLASS

Exclude a class if it has the **NoInstallClass** value entry in its registry key.

DIBCI_NODISPLAYCLASS

Exclude a class if it has the **NoDisplayClass** value entry in its registry key.

ClassGuidList

Supplies a pointer to a buffer that receives a list of setup class GUIDs.

ClassGuidListSize

Supplies the number of GUIDs in the *ClassGuildList* array.

RequiredSize

Supplies a pointer to a variable that receives the number of GUIDs returned. If this number is greater than the size of the *ClassGuidList*, the number indicates how large the *ClassGuid-List* array must be in order to contain the list.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

To retrieve the list of setup class GUIDs installed on a remote system use **SetupDiBuild-ClassInfoListEx**.

See Also

SetupDiBuildClassInfoListEx, **SetupDiGetClassDescription**, **SetupDiGetINFClass**

SetupDiBuildClassInfoListEx

```
BOOLEAN
  SetupDiBuildClassInfoListEx(
    IN DWORD  Flags,
    OUT LPGUID  ClassGuidList,
    IN DWORD  ClassGuidListSize,
    OUT PDWORD  RequiredSize,
    IN PCTSTR  MachineName,  OPTIONAL
    IN PVOID  Reserved,
    );
```

SetupDiBuildClassInfoListEx returns a list of setup class GUIDs that includes every class installed on the local system or a remote system.

Parameters

Flags

Flags used to control exclusion of classes from the list. If no flags are specified, all setup classes are included in the list. Can be a combination of the following values:

DIBCI_NOINSTALLCLASS

Exclude a class if it has the **NoInstallClass** value entry in its registry key.

DIBCI_NODISPLAYCLASS

Exclude a class if it has the **NoDisplayClass** value entry in its registry key.

ClassGuidList

Supplies a pointer to a buffer that receives a list of setup class GUIDs.

ClassGuidListSize

Supplies the number of GUIDs in the *ClassGuildList* array.

RequiredSize

Supplies a pointer to a variable that receives the number of GUIDs returned. If this number is greater than the size of the *ClassGuidList*, the number indicates how large the *ClassGuid-List* array must be in order to contain the list.

MachineName

Optionally supplies the name of a remote machine from which to retrieve installed setup classes. If *MachineName* is NULL, this function builds a list of classes installed on the local machine.

Reserved

Must be NULL.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

See Also

SetupDiBuildClassInfoList, **SetupDiGetClassDescriptionEx**

SetupDiBuildDriverInfoList

```
BOOLEAN
  SetupDiBuildDriverInfoList(
    IN HDEVINFO DeviceInfoSet,
    IN OUT PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN DWORD  DriverType
    );
```

SetupDiBuildDriverInfoList builds a list of drivers associated with a specified device instance or with the device information set's global class driver list.

Parameters

DeviceInfoSet

Supplies a handle to the device information set to contain the driver information list (either globally for all elements or specifically for a single element). The device information set must not contain remote elements.

DeviceInfoData

Supplies a pointer to the SP_DEVINFO_DATA structure for the device information element for which to build a driver list. If this parameter is NULL, the list is associated with the device information set and not with a particular device information element. Use NULL with driver lists of type SPDIT_CLASSDRIVER only.

If the class of this device is updated as a result of building a compatible driver list, the **ClassGuid** field of the SP_DEVINFO_DATA structure is updated upon return.

DriverType

Specifies what type of driver list to build. Must be one of the following values:

SPDIT_CLASSDRIVER

Build a list of class drivers.

SPDIT_COMPATDRIVER

Build a list of drivers for this device. *DeviceInfoData* must be specified if this flag is set.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

The caller can set **Flags** in the SP_DEVINSTALL_PARAMS that are associated with the device information set or with a specific device (*DeviceInfoData*) to control how the list is built. For example, the caller can set the DI_FLAGSEX_ALLOWEXCLUDEDDRVS flag to include drivers that are marked Exclude From Select.

A driver is Exclude From Select if either it is marked **ExcludeFromSelect** in the INF file or it is a driver for a device whose whole setup class is marked **NoInstallClass** or **NoUseClass** in the class installer INF. Drivers for PnP devices are typically "Exclude From Select"; PnP devices should not be manually installed. To build a list of driver files for a PnP device a caller of **SetupDiBuildDriverInfoList** must set this flag.

The **DriverPath** in the SP_DEVINSTALL_PARAMS contains either a path to a directory containing INF files or a path to a specific INF file. If DI_ENUMSINGLEINF is set, **DriverPath** contains a path to a single INF file. If **DriverPath** is NULL, this function builds the driver list from the default INF location, %windir%\inf.

After this function has built the specified driver list, the caller can enumerate the elements of the list by calling **SetupDiEnumDriverInfo**.

If the driver list is associated with a device instance (that is, *DeviceInfoData* is specified), the resulting list is composed of drivers that have the same class as the device instance with which they are associated. If this is a global class driver list (that is, *DriverType* is SPDIT_CLASSDRIVER and *DeviceInfoData* is not specified), the class that is used when building the list is the class associated with the device information set. If the device information set has no associated class, drivers of all classes are used when building the list.

Another thread can terminate the building of a driver list by a call to **SetupDiCancel-DriverInfoSearch**.

The *DeviceInfoSet* must only contain elements on the local machine. This function only searches for local drivers.

See Also

SetupDiCancelDriverInfoSearch, **SetupDiDestroyDriverInfoList**, **SetupDiEnum-DriverInfo**

SetupDiCallClassInstaller

```
BOOLEAN
  SetupDiCallClassInstaller(
    IN DI_FUNCTION  InstallFunction,
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
    );
```

SetupDiCallClassInstaller calls any registered coinstallers and the appropriate class installer with the specified installation request (DIF code).

Parameters

InstallFunction

Specifies the device installation request (DIF request) to pass to the coinstallers and class installer. DIF codes have the format DIF_*XXX* and are defined in *setupapi.h*. See *Device Installation Function Codes* for more information.

DeviceInfoSet

Supplies a handle to a device information set for the local machine.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that specifies a device in the *DeviceInfoSet*. If *DeviceInfoData* is NULL, this function calls the installers for the setup class associated with the device information set.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

SetupDiCallClassInstaller calls the class installer and any coinstallers that are registered for a device or a device setup class. This function loads the installers if they are not yet loaded. This function also calls the default handler for the DIF request, if there is a default handler and if the installers returned a status indicating that the default handler should be called.

Setup applications call this function with a variety of device installation function requests (DIF requests). This function ensures that all the appropriate installers and default handlers are called, in the correct order, for a given DIF request.

The *DeviceInfoSet* must only contain elements on the local machine.

See the *Plug and Play, Power Management, and Setup Design Guide* for information on the design and operation of class installers and coinstallers.

See Also

SP_DEVINFO_DATA

SetupDiCancelDriverInfoSearch

```
BOOLEAN
  SetupDiCancelDriverInfoSearch(
    IN HDEVINFO DeviceInfoSet
    );
```

SetupDiCancelDriverInfoSearch cancels a driver list search that is currently underway in a different thread.

Parameters

DeviceInfoSet

Supplies a handle to the device information set for which a driver list is being built.

Return Value

If a driver list search is underway for the specified device information set when this function is called, the search is terminated. **SetupDiCancelDriverInfoSearch** returns TRUE when the termination is confirmed. Otherwise it returns FALSE and a call to **GetLastError** returns ERROR_INVALID_HANDLE.

Comments

SetupDiCancelDriverInfoSearch is a synchronous call; therefore, it does not return until the driver search thread responds to the termination request.

See Also

SetupDIBuildDriverInfoList

SetupDiChangeState

```
BOOLEAN
  SetupDiChangeState(
    IN HDEVINFO  DeviceInfoSet,
    IN OUT PSP_DEVINFO_DATA  DeviceInfoData
    );
```

SetupDiChangeState is the default handler for the DIF_PROPERTYCHANGE installation request. This function changes the state of an installed device.

Parameters

DeviceInfoSet

Supplies a handle to a device information set for the local machine.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that specifies a device in the *DeviceInfoSet.* This is an IN OUT parameter because the **DevInst** field of the structure can be updated with a new handle value upon return.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

Callers of this function must specify a DICS_*XXX* flag in the SP_PROPCHANGE_PARAMS for the device element that indicates the type of state change to perform on the device. Callers of this function must set the appropriate fields in the SP_PROPCHANGE_PARAMS and call **SetupDiSetClassInstallParams** before calling this function.

If you specify the DICS_FLAG_CONFIGSPECIFIC flag in the SP_PROPCHANGE_PARAMS then you must fill in the **HwProfile** field. A value of zero for **HwProfile** indicates the current profile.

To enable/disable a device in the current hardware profile, set the DICS_FLAG_CONFIGSPECIFIC flag in the SP_PROPCHANGE_PARAMS. To enable/disable a device globally, such as in both the docked and undocked hardware profiles, set the DICS_FLAG_GLOBAL flag.

This function does the following:

DICS_ENABLE

Loads the drivers for the device and starts the device, if possible. If the function is not able to start the device, it sets the DI_NEEDREBOOT flag for the device which indicates to the initiator of the property change request that they must prompt the user to reboot the machine.

DICS_DISABLE

Disables the device. If the device is disableable but this function cannot disable the device dynamically, this function marks the device to be disabled the next time the machine reboots.

DICS_PROPCHANGE

Removes and reconfigures the device so the new properties can take effect. This flag typically indicates that a user has changed a property on a Device Manager property page for the device. The PnP Manager directs the drivers for the device to remove their device objects and then the PnP Manager reconfigures and restarts the device.

Callers of this function should not specify DICS_STOP or DICS_START in the SP_PROPCHANGE_PARAMS. Use DICS_PROPCHANGE to stop and restart a device to cause changes in the device's configuration to take effect.

If DI_DONOTCALLCONFIGMG is set for a device, you should not call **SetupDiChange-State** for the device but should instead set the DI_NEEDREBOOT flag.

See Also

DIF_PROPERTYCHANGE, **SetupDiCallClassInstaller**, SP_PROPCHANGE_PARAMS

SetupDiClassGuidsFromName

```
BOOLEAN
  SetupDiClassGuidsFromName(
    IN PCTSTR  ClassName,
    OUT LPGUID  ClassGuidList,
    IN DWORD  ClassGuidListSize,
    OUT PDWORD  RequiredSize
    );
```

SetupDiClassGuidsFromName retrieves the GUID(s) associated with the specified class name. This list is built based on the classes currently installed on the system.

Parameters

ClassName

Supplies the name of the class for which to retrieve the class GUID.

ClassGuidList

Supplies a pointer to an array to receive the list of GUIDs associated with the specified class name.

ClassGuidListSize

Supplies the number of GUIDs in the *ClassGuidList* array.

RequiredSize

Supplies a pointer to a variable that receives the number of GUIDs associated with the class name. If this number is greater than the size of the *ClassGuidList* buffer, the number indicates how large the array must be in order to store all the GUIDs.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

Call **SetupDiClassGuidsFromNameEx** to retrieve the class GUIDs for a class on a remote machine.

See Also

SetupDiClassGuidsFromNameEx, **SetupDiClassNameFromGuid**

SetupDiClassGuidsFromNameEx

```
BOOLEAN
  SetupDiClassGuidsFromNameEx(
    IN PCTSTR  ClassName,
    OUT LPGUID  ClassGuidList,
    IN DWORD  ClassGuidListSize,
    OUT PDWORD  RequiredSize,
    IN PCSTR  MachineName,  OPTIONAL
    IN PVOID  Reserved
    );
```

SetupDiClassGuidsFromNameEx retrieves the GUID(s) associated with the specified class name. This resulting list contains the classes currently installed on a local or remote machine.

Parameters

ClassName

Supplies the name of the class for which to retrieve the class GUIDs.

ClassGuidList

Supplies a pointer to an array to receive the list of GUIDs associated with the specified class name.

ClassGuidListSize

Supplies the number of GUIDs in the *ClassGuidList* array.

RequiredSize

Supplies a pointer to a variable that receives the number of GUIDs associated with the class name. If this number is greater than the size of the *ClassGuidList* buffer, the number indicates how large the array must be in order to store all the GUIDs.

MachineName

Optionally supplies the name of a remote machine from which to retrieve the GUIDs. If *MachineName* is NULL, the local machine is used.

Reserved

Must be NULL.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

Class names are not guaranteed to be unique; only GUIDs are unique. Therefore, one class name can return more than one GUID.

See Also

SetupDiClassGuidsFromName, **SetupDiClassNameFromGuidEx**

SetupDiClassNameFromGuid

```
BOOLEAN
  SetupDiClassNameFromGuid(
    IN LPGUID  ClassGuid,
    OUT PTSTR  ClassName,
    IN DWORD  ClassNameSize,
    OUT PDWORD  RequiredSize  OPTIONAL
    );
```

SetupDiClassNameFromGuid retrieves the class name associated with a class GUID.

Parameters

ClassGuid

Supplies the class GUID of the class name to retrieve.

ClassName

Receives the name of the class for the specified GUID.

ClassNameSize

Supplies the size, in characters, of the *ClassName* buffer.

RequiredSize

Receives the number of characters required to store the class name (including terminating NULL). *RequiredSize* is always less than MAX_CLASS_NAME_LEN.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

Call **SetupDiClassNameFromGuidEx** to retrieve the name for a class on a remote machine.

See Also

SetupDiClassGuidsFromName,.**SetupDiClassNameFromGuidEx**

SetupDiClassNameFromGuidEx

```
BOOLEAN
  SetupDiClassNameFromGuidEx(
    IN LPGUID  ClassGuid,
    OUT PTSTR  ClassName,
    IN DWORD  ClassNameSize,
    OUT PDWORD  RequiredSize,  OPTIONAL
    IN PCSTR  MachineName,  OPTIONAL
    IN PVOID  Reserved
    );
```

SetupDiClassNameFromGuidEx retrieves the class name associated with a class GUID. The class can be installed on a local or remote machine.

Parameters

ClassGuid

Supplies the class GUID of the class name to retrieve.

ClassName

Receives the name of the class for the specified GUID.

ClassNameSize

Supplies the size, in characters, of the *ClassName* buffer.

RequiredSize

Receives the number of characters required to store the class name (including terminating NULL). *RequiredSize* is always less than MAX_CLASS_NAME_LEN.

MachineName

Optionally supplies the name of a remote machine on which the class is installed. If *MachineName* is NULL, the local machine is used.

Reserved

Must be NULL.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

See Also

SetupDiClassGuidsFromNameEx,.**SetupDiClassNameFromGuid**

SetupDiCreateDeviceInfo

```
BOOLEAN
  SetupDiCreateDeviceInfo(
    IN HDEVINFO  DeviceInfoSet,
    IN PCTSTR  DeviceName,
    IN LPGUID  ClassGuid,
    IN PCTSTR  DeviceDescription,  OPTIONAL
    IN HWND  hwndParent,  OPTIONAL
    IN DWORD  CreationFlags,
    OUT PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
    );
```

SetupDiCreateDeviceInfo creates a new device information element and adds it as a new member to the specified device information set.

Parameters

DeviceInfoSet

Supplies a handle to the device information set for the local machine.

DeviceName

Supplies either a full device instance ID (for example, **Root*PNP0500\0000**) or a root-enumerated device ID without the **Enum** branch prefix and instance ID suffix (for example, ***PNP0500**). The root-enumerated device ID can be used only if the DICD_GENERATE_ID flag is specified in the *CreationFlags* parameter.

ClassGuid

Supplies a pointer to the GUID for this device's class. If the class is not known, this value should be GUID_NULL.

DeviceDescription

Supplies a textual description of the device.

hwndParent

Supplies the window handle of the top-level window to use for any user interface related to installing the device.

CreationFlags

Controls how the device information element is created. Can be a combination of the following values:

DICD_GENERATE_ID

If this flag is specified, *DeviceName* contains only a Root-enumerated device ID and the system creates a unique device instance key for it. This unique device instance key is generated as:

Enum\Root\*DeviceName**InstanceID* where *InstanceID* is a four-digit, base-10 number that is unique among all subkeys under **Enum\Root\***DeviceName*. Call **SetupDiGetDeviceInstanceId** to find out what ID was generated for this device information element.

DICD_INHERIT_CLASSDRVS

If this flag is specified, the resulting device information element inherits the class driver list, if any, associated with the device information set. In addition, if there is a selected driver for the device information set, that same driver is selected for the new device information element.

DeviceInfoData

Optionally supplies a pointer to an SP_DEVINFO_DATA structure to receive the new device information element. If this parameter is not NULL, the caller must set **cbSize** to **sizeof**(SP_DEVINFO_DATA).

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

If this device instance is being added to a set that has an associated class, the device class must be the same or the call fails. In this case, a call to **GetLastError** returns ERROR_CLASS_MISMATCH.

If the specified device instance is the same as an existing device instance key in the registry, the call fails. In this case, a call to **GetLastError** returns ERROR_DEVINST_ALREADY_EXISTS. This occurs only if the DCID_GENERATE_ID flag is not set.

If the new device information element was successfully created but the caller-supplied *DeviceInfoData* buffer is invalid, the function returns FALSE. In this case, a call to **GetLastError** returns ERROR_INVALID_USER_BUFFER. However, the device information element will have been added as a new member of the set already.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiDeleteDeviceInfo, **SetupDiEnumDeviceInfo**, **SetupDiOpenDeviceInfo**, SP_DEVINFO_DATA

SetupDiCreateDeviceInfoList

```
HDEVINFO
  SetupDiCreateDeviceInfoList(
    IN LPGUID  ClassGuid,  OPTIONAL
    IN HWND  hwndParent  OPTIONAL
    );
```

SetupDiCreateDeviceInfoList creates an empty device information set. This set can be associated with a class GUID.

Parameters

ClassGuid

Optionally supplies the GUID of the setup class associated with this device information set. If this parameter is specified, only devices of this class may be included in this device information set.

hwndParent

Optionally supplies the window handle of the top-level window to use for any user interface related to non-device-specific actions (such as a select-device dialog using the global class driver list).

Return Value

The function returns a handle to an empty device information set if it is successful. Otherwise it returns INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**.

Comments

The caller of this function must delete the returned device information set when it is no longer needed by calling **SetupDiDestroyDeviceInfoList**.

To create a device information list for a remote machine use **SetupDiCreateDeviceInfoListEx**.

See Also

SetupDiCreateDeviceInfoListEx, **SetupDiGetClassDevs**, **SetupDiDestroyDeviceInfoList**, **SetupDiGetDeviceInfoListClass**

SetupDiCreateDeviceInfoListEx

```
HDEVINFO
  SetupDiCreateDeviceInfoListEx(
    IN LPGUID  ClassGuid,  OPTIONAL
    IN HWND  hwndParent,  OPTIONAL
    IN PCTSTR  MachineName,  OPTIONAL
    IN PVOID  Reserved
    );
```

SetupDiCreateDeviceInfoListEx creates an empty device information set. This set can be associated with a class GUID and can be for devices on a remote machine.

Parameters

ClassGuid

Optionally supplies the GUID of the setup class associated with this device information set. If this parameter is specified, only devices of this class may be included in this device information set.

hwndParent

Optionally supplies the window handle of the top-level window to use for any user interface related to non-device-specific actions (such as a select-device dialog using the global class driver list).

MachineName

Optionally supplies the name of a machine on the network. If such a name is specified, only devices on that machine can be created and opened in this device information set. If *MachineName* is NULL, the device information set is for local devices.

Reserved

Must be NULL.

Return Value

The function returns a handle to an empty device information set if it is successful. Otherwise, it returns INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**.

Comments

The caller of this function must delete the returned device information set when it is no longer needed by calling **SetupDiDestroyDeviceInfoList**.

If the device information set is for devices on a remote machine (*MachineName* is not NULL), all subsequent operations on this set or any of its elements must use routines that support device information sets with remote elements. The **SetupDi***Xxx* routines that do not provide this support, such as **SetupDiCallClassInstaller**, have a statement to that effect in their reference page.

See Also

SetupDiCreateDeviceInfoList, **SetupDiDestroyDeviceInfoList**, **SetupDiGetDeviceInfoListDetail**

SetupDiCreateDeviceInterface

```
BOOLEAN
  SetupDiCreateDeviceInterface(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,
    IN LPGUID  InterfaceClassGuid,
    IN PCTSTR  ReferenceString,  OPTIONAL
    IN DWORD  CreationFlags,
    OUT PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData  OPTIONAL
    );
```

SetupDiCreateDeviceInterface registers device functionality (a device interface) for a device.

Parameters

DeviceInfoSet

Points to the device information set containing the device for which an interface is being registered. This handle is typically returned by **SetupDiGetClassDevs**. The device information set must not contain remote elements.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

InterfaceClassGuid

Points to a class GUID that specifies the interface class for the new interface.

ReferenceString

Optionally points to a reference string; this parameter is typically NULL.

Reference strings are only used by a few bus drivers that use device interfaces as placeholders for software devices that are created on demand.

CreationFlags

Reserved. Must be zero.

DeviceInterfaceData

Optionally points to a caller-allocated buffer to receive information about the new device interface. The caller must set *DeviceInterfaceData*.**cbSize** to **sizeof**(SP_DEVICE_INTERFACE_DATA) before calling this function.

Return Value

SetupDiCreateDeviceInterface returns TRUE if the function completed without error. If the function completed with an erorr, it returns FALSE and the error code for the failure can be retrieved by calling **GetLastError**.

Comments

SetupDiCreateDeviceInterface registers an interface for a device. If a device has more than one interface, call this function once for each interface being registered.

Before a registered interface can be used by applications and other system components the interface must be enabled by the driver for the device.

This function creates a registry key for the new device interface. Callers of this function can access non-volatile storage under this key using **SetupDiOpenDeviceInterfaceRegKey**.

If the new device interface is successfully created and registered, but the caller-supplied *DeviceInterfaceData* buffer is invalid, this function returns FALSE and **GetLastError** returns ERROR_INVALID_USER_BUFFER. The caller's buffer error does not prevent the interface from being created and registered.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiOpenDeviceInterfaceRegKey, **SetupDiRemoveDeviceInterface**

SetupDiCreateDeviceInterfaceRegKey

```
HKEY
  SetupDiCreateDeviceInterfaceRegKey(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
    IN DWORD  Reserved,
    IN REGSAM  samDesired,
    IN HINF  InfHandle,  OPTIONAL
    IN PCTSTR  InfSectionName  OPTIONAL
    );
```

SetupDiCreateDeviceInterfaceRegKey creates a registry key for storing information about a device interface instance and returns a handle to the key.

Parameters

DeviceInfoSet

Points to a device information set containing the interface and its underlying device. The device information set must not contain remote elements.

DeviceInterfaceData

Points to a structure that identifies the device interface, possibly returned by **SetupDi-CreateDeviceInterface**.

Reserved

Reserved. Must be zero.

samDesired

Specifies the registry access requested by the caller to the key being created.

InfHandle

Optionally supplies the handle of an open INF file that contains a *DDInstall* section to be executed for the newly-created key. If this parameter is not NULL, *InfSectionName* must be specified as well.

InfSectionName

Optionally points to the name of an *DDInstall* section in the INF file specified by *InfHandle*. This section is executed for the newly created key. If this parameter is not NULL, *InfHandle* must be specified as well.

Return Value

SetupDiCreateDeviceInterfaceRegKey returns a handle to the newly-created registry key. If the function fails, it returns INVALID_HANDLE_VALUE. Call **GetLastError** to get extended error information.

Comments

SetupDiCreateDeviceInterfaceRegKey creates a non-volatile subkey of the registry key for the specified device interface. Callers of this function can store private configuration data for the device interface in this subkey. The driver for the device can access this subkey using **Io***Xxx* routines.

Close the handle returned from this function by calling **RegCloseKey**.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiCreateDeviceInterface, **SetupDiDeleteDeviceInterfaceRegKey**, **SetupDiOpen-DeviceInterfaceRegKey**

SetupDiCreateDevRegKey

```
HKEY
  SetupDiCreateDevRegKey(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,
    IN DWORD  Scope,
    IN DWORD  HwProfile,
    IN DWORD  KeyType,
    IN HINF  InfHandle,  OPTIONAL
    IN PCTSTR  InfSectionName  OPTIONAL
    );
```

SetupDiCreateDevRegKey creates a registry storage key for device-specific configuration information and returns a handle to the key.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing information about the device instance whose registry configuration storage key is to be created. The device information set must not contain remote elements.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure indicating the device instance for which to create the registry key.

Scope

Specifies the scope of the registry key to be created. The scope determines where the information is stored. The key created can be global or hardware profile-specific. Can be one of the following values:

DICS_FLAG_GLOBAL

Create a key to store global configuration information. This information is not specific to a particular hardware profile. On Windows NT®/Windows 2000 this creates a key that is rooted at **HKEY_LOCAL_MACHINE.** The exact key opened depends on the value of the *KeyType* parameter.

DICS_FLAG_CONFIGSPECIFIC

Create a key to store hardware profile-specific configuration information. This key is rooted at one of the hardware-profile specific branches, instead of **HKEY_LOCAL_MACHINE**.

HwProfile

Specifies the hardware profile for which to create a key if *HwProfileFlags* is set to SPDICS_FLAG_CONFIGSPECIFIC. If *HwProfile* is 0, the key for the current hardware profile is created. If *HwProfileFlags* is SPDICS_FLAG_GLOBAL, *HwProfile* is ignored.

KeyType

Specifies the type of registry storage key to create. Can be one of the following values:

DIREG_DEV

Create a hardware registry key for the device. This is the key for storage of driver-independent configuration information. This key is in the *DeviceInstance* key of the **Enum** branch.

DIREG_DRV

Create a software, or driver, registry key for the device. This key is located in the **Class** branch.

InfHandle

Supplies the handle of an open INF file that contains a *DDInstall* section to be executed for the newly-created key. If this parameter is specified, *InfSectionName* must be specified as well.

InfSectionName

Supplies the name of a *DDInstall* section in the INF file specified by *InfHandle*. This section is executed for the newly created key. If this parameter is specified, *InfHandle* must be specified as well.

Return Value

If the function is successful, it returns a handle to the newly-created registry key where private configuration data pertaining to this device instance can be stored/retrieved. If the function fails, it returns INVALID_HANDLE_VALUE. Call **GetLastError** to get extended error information.

Comments

Close the handle returned from this function by calling **RegCloseKey**.

The specified device instance must be registered before calling this function. After creating the device instance with **SetupDiCreateDeviceInfo**, call **SetupDiRegisterDeviceInfo** to register it.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiCreateDeviceInfo, **SetupDiGetHwProfileList**, **SetupDiOpenDevRegKey**, **SetupDiRegisterDeviceInfo**

SetupDiDeleteDeviceInfo

```
BOOLEAN
  SetupDiDeleteDeviceInfo(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData
    );
```

SetupDiDeleteDeviceInfo deletes a member from the specified device information set. This function does not delete the actual device.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing the device information member to delete.

DeviceInfoData

Supplies a pointer to the SP_DEVINFO_DATA structure for the device information member to delete.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

If the specified device information element is in use (for example, by a wizard page), the function fails. In this case, a call to **GetLastError** returns ERROR_DEVINFO_DATA_LOCKED. This happens if a handle to a wizard page is retrieved with a call to **SetupDiGetWizardPage** with this device information element specified and the DIWP_FLAG_USE_DEVINFO_DATA flag set. To delete this device information element, you must first close the wizard's HPROPSHEETPAGE handle.

See Also

SetupDiCreateDeviceInfo, **SetupDiEnumDeviceInfo**, **SetupDiGetWizardPage**, **SetupDiOpenDeviceInfo**

SetupDiDeleteDeviceInterfaceData

```
BOOLEAN
  SetupDiDeleteDeviceInterfaceData(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData
    );
```

SetupDiDeleteDeviceInterfaceData deletes a device interface from a device information set.

Parameters

DeviceInfoSet

Points to the device information set containing the interface and its underlying device. This handle is typically returned by **SetupDiGetClassDevs**.

DeviceInterfaceData

Points to a structure that identifes the interface to be deleted. This structure is typically returned by **SetupDiEnumDeviceInterfaces**.

Return Value

SetupDiDeleteDeviceInterfaceData returns TRUE if the function completed without error. If the function completed with an erorr, it returns FALSE and the error code for the failure can be retrieved by calling **GetLastError**.

Comments

SetupDiDeleteDeviceInterfaceData deletes a device interface element from a device information set. This function has no effect on the device interface or the underlying device.

See Also

SetupDiEnumDeviceInterfaces, **SetupDiGetClassDevs**, **SetupDiRemoveDeviceInterface**

SetupDiDeleteDeviceInterfaceRegKey

```
BOOLEAN
  SetupDiDeleteDeviceInterfaceRegKey(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
    IN DWORD  Reserved
    );
```

SetupDiDeleteDeviceInterfaceRegKey deletes the registry subkey that was used by applications and drivers to store information specific to a device interface instance.

Parameters

DeviceInfoSet

Points to a device information set containing the interface and its underlying device. The device information set must not contain remote elements.

DeviceInterfaceData

Points to a structure that identifies the device interface, possibly returned by **SetupDi-CreateDeviceInterface** or **SetupDiEnumDeviceInterfaces**.

Reserved

Reserved. Must be zero.

Return Value

SetupDiDeleteDeviceInterfaceRegKey returns TRUE if it is successful; otherwise, it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

SetupDiDeleteDeviceInterfaceRegKey deletes the subkey used by drivers and applications to store information about the device interface instance. This subkey was created by **SetupDiCreateDeviceInterfaceRegKey** or by the driver's call to an associated **Io***Xxx* routine. **SetupDiDeleteDeviceInterfaceRegKey** does not affect the main registry key for the device interface instance nor any other subkeys that may have been created.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiCreateDeviceInterface, **SetupDiCreateDeviceInterfaceRegKey**

SetupDiDeleteDevRegKey

```
BOOLEAN
  SetupDiDeleteDevRegKey(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,
    IN DWORD  Scope,
    IN DWORD  HwProfile,
    IN DWORD  KeyType
    );
```

SetupDiDeleteDevRegKey deletes the specified user-accessible registry key(s) associated with a device information element.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing the device instance whose registry configuration storage key is to be deleted. The device information set must not contain remote elements.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure indicating the device instance for which to delete registry key(s).

Scope

Specifies the scope of the registry key to delete. The scope indicates where the information is located. The key can be global or hardware profile-specific. Can be one of the following values:

DICS_FLAG_GLOBAL

Delete the key that stores global configuration information.

DICS_FLAG_CONFIGSPECIFIC

Delete the key that stores hardware profile-specific configuration information.

HwProfile

If *Scope* is set to DICS_FLAG_CONFIGSPECIFIC, the *HwProfile* parameter specifies the hardware profile for which to delete the registry key. If *HwProfile* is 0, the key for the current hardware profile is deleted. If *HwProfile* is 0xFFFFFFFF, the registry key for all hardware profiles is deleted.

KeyType

Specifies the type of registry storage key to delete. Can be one of the following values:

DIREG_DEV

Delete the hardware registry key for the device. This is the key for storage of driver-independent configuration information. This key is in the **Device Instance** key of the **Enum** branch.

DIREG_DRV

Delete a software, or driver, registry key for the device. This key is located in the **Class** branch.

DIREG_BOTH

Delete both the hardware and software keys for the device.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiCreateDevRegKey, SetupDiGetHwProfileList

SetupDiDestroyClassImageList

```
BOOLEAN
  SetupDiDestroyClassImageList(
    IN PSP_CLASSIMAGELIST_DATA  ClassImageListData
    );
```

SetupDiDestroyClassImageList destroys a class image list that was built by a call to **SetupDiGetClassImageList** or **SetupDiGetClassImageListEx**.

Parameters

ClassImageListData

Supplies a pointer to an SP_CLASSIMAGELIST_DATA structure that contains the class image list to destroy.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

See Also

SetupDiGetClassImageList, **SetupDiGetClassImageListEx**

SetupDiDestroyDeviceInfoList

```
BOOLEAN
  SetupDiDestroyDeviceInfoList(
    IN HDEVINFO  DeviceInfoSet
    );
```

SetupDiDestroyDeviceInfoList destroys a device information set and frees all associated memory.

Parameters

DeviceInfoSet

Supplies a handle to the device information set to destroy.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

See Also

SetupDiCreateDeviceInfoList, **SetupDiGetClassDevs**

SetupDiDestroyDriverInfoList

```
BOOLEAN
  SetupDiDestroyDriverInfoList(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN DWORD  DriverType
    );
```

SetupDiDestroyDriverInfoList destroys a driver information list.

Parameters

DeviceInfoSet

Supplies a handle to the device information set that contains the driver information list to destroy.

DeviceInfoData

Supplies a pointer to the SP_DEVINFO_DATA structure associated with the driver information list to destroy. If this parameter is not specified, the global class driver list is destroyed.

DriverType

Specifies what type of driver list to destroy. Must be one of the following values:

SPDIT_CLASSDRIVER

Destroy a list of class drivers.

SPDIT_COMPATDRIVER

Destroy a list of compatible drivers for the specified device. *DeviceInfoData* must be specified if this flag is set.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

If the currently selected driver is a member of the list being destroyed, the selection is reset.

If a class driver list is being destroyed, the DI_FLAGSEX_DIDINFOLIST and DI_DIDCLASS flags are reset for the corresponding device information set or device information element. The DI_MULTMFGS flags is also reset.

If a compatible driver list is being destroyed, the DI_FLAGSEX_DIDCOMPATINFO and DI_DIDCOMPAT flags are reset for the corresponding device information element.

See Also

SetupDiBuildDriverInfoList

SetupDiDrawMiniIcon

```
INT
  SetupDiDrawMiniIcon(
    IN HDC  hdc,
    IN RECT  rc,
    IN INT  MiniIconIndex,
    IN DWORD  Flags
    );
```

SetupDiDrawMiniIcon draws the specified mini-icon at the location requested.

Parameters

hdc

Supplies the handle of the device context in which the mini-icon will be drawn.

rc

The rectangle in the specified device context handle to draw the mini-icon in.

MiniIconIndex

The index of the mini-icon, as retrieved from **SetupDiLoadClassIcon** or **SetupDiGetClassBitmapIndex**. The following predefined indexes for devices can be used:

Class	Index
Computer	0
Display	2
Mouse	5
Keyboard	6
FDC	9
HDC	9
Ports	10

Continued

Class	Index
Net	15
System	0
Sound	8
Printer	14
Monitor	2
Network Transport	3
Network Client	16
Network Service	17
Unknown	18

Flags

These flags control the drawing operation. The LOWORD contains the actual flags defined as follows:

DMI_MASK

Draw the mini-icon's mask into HDC.

DMI_BKCOLOR

Use the system color index specified in the HIWORD of *Flags* as the background color. If this flag is not set, COLOR_WINDOW is used.

DMI_USERECT

If set, **SetupDiDrawMiniIcon** uses the supplied rectangle and stretches the icon to fit.

Return Value

This function returns the offset from the left side of *rc* where the string should start.

Comments

By default, the icon is centered vertically and butted against the left side of the specified rectangle.

See Also

SetupDiGetClassBitmapIndex, **SetupDiLoadClassIcon**

SetupDiEnumDeviceInfo

```
BOOLEAN
  SetupDiEnumDeviceInfo(
    IN HDEVINFO  DeviceInfoSet,
    IN DWORD  MemberIndex,
    OUT PSP_DEVINFO_DATA  DeviceInfoData
    );
```

SetupDiEnumDeviceInfo returns a context structure for a device information element of a device information set. Each call returns information about one device; the function can be called repeatedly to get information about several devices.

Parameters

DeviceInfoSet

Supplies a handle to the device information set.

MemberIndex

Supplies the zero-based index of the device information element to retrieve.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure to receive information about this element. The caller must set **cbSize** to **sizeof**(SP_DEVINFO_DATA).

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

To enumerate device information elements, an installer should initially call **SetupDiEnumDeviceInfo** with the *MemberIndex* parameter set to 0. The installer should then increment *MemberIndex* and call **SetupDiEnumDeviceInfo** until there are no more values (the function fails and a call to **GetLastError** returns ERROR_NO_MORE_ITEMS).

Call **SetupDiEnumDeviceInterfaces** to get a context structure for a device *interface* element (vs. a device *information* element).

See Also

SetupDiCreateDeviceInfo, **SetupDiDeleteDeviceInfo**, **SetupDiEnumDeviceInterfaces**, **SetupDiOpenDeviceInfo**,SP_DEVINFO_DATA

SetupDiEnumDeviceInterfaces

```
BOOLEAN
  SetupDiEnumDeviceInterfaces(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN LPGUID  InterfaceClassGuid,
    IN DWORD  MemberIndex,
    OUT PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData
    );
```

SetupDiEnumDeviceInterfaces returns a context structure for a device interface element of a device information set. Each call returns information about one device interface; the function can be called repeatedly to get information about several interfaces exposed by one or more devices.

Parameters

DeviceInfoSet

Points to a device information set containing the device(s) for which to return interface information. This handle is typically returned by **SetupDiGetClassDevs**.

DeviceInfoData

Optionally points to an SP_DEVINFO_DATA structure that constrains the search for interfaces to those of just one device in the device information set. This pointer is typically returned by **SetupDiEnumDeviceInfo**.

InterfaceClassGuid

Points to a GUID that specifies the device interface class for the requested interface.

MemberIndex

Specifies a zero-based index into the list of interfaces in the device information set. The caller should call this function first with *MemberIndex* set to zero to obtain the first interface. Then, repeatedly increment *MemberIndex* and retrieve an interface until this function fails and **GetLastError** returns ERROR_NO_MORE_ITEMS.

If *DeviceInfoData* specifies a particular device, the *MemberIndex* is relative to only the interfaces exposed by that device.

DeviceInterfaceData

Points to a caller-allocated buffer that contains, on successful return, a completed SP_DEVICE_INTERFACE_DATA structure that identifies an interface that meets the search parameters. The caller must set *DeviceInterfaceData*.**cbSize** to **sizeof**(SP_DEVICE_INTERFACE_DATA) before calling this function.

Return Value

SetupDiEnumDeviceInterfaces returns TRUE if the function completed without error. If the function completed with an error, FALSE is returned and the error code for the failure can be retrieved by calling **GetLastError**.

Comments

DeviceInterfaceData points to a structure that identifies a requested device interface. To get detailed information about an interface, call **SetupDiGetDeviceInterfaceDetail**. The detailed information includes the name of the device interface that can be passed to a Win32® function such as **CreateFile** to get a handle to the interface.

See Also

SetupDiGetClassDevs, **SetupDiEnumDeviceInfo**, **SetupDiGetDeviceInterfaceDetail**

SetupDiEnumDriverInfo

```
BOOLEAN
  SetupDiEnumDriverInfo(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN DWORD  DriverType,

 IN DWORD  MemberIndex,

OUT PSP_DRVINFO_DATA  DriverInfoData
    );
```

SetupDiEnumDriverInfo enumerates the members of a driver information list.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing a driver information list to enumerate.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that contains a driver information list to enumerate. If this parameter is not specified, the global driver list owned by the device information set is used (this list is of type SPDIT_CLASSDRIVER).

DriverType

Specifies what type of driver list to enumerate. Must be one of the following values:

SPDIT_CLASSDRIVER

Enumerate a class driver list.

SPDIT_COMPATDRIVER

Enumerate a list of compatible drivers for the specified device. *DeviceInfoData* must be specified if this flag is set.

MemberIndex

Supplies the 0-based index of the driver information member to retrieve.

DriverInfoData

Supplies a pointer to an SP_DRVINFO_DATA structure to receive information about the enumerated driver.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

To enumerate driver information set members, an installer should first call **SetupDi-EnumDriverInfo** with the *MemberIndex* parameter set to 0. It should then increment *MemberIndex* and call **SetupDiEnumDriverInfo** until there are no more values. When there are no more values, the function fails and a call to **GetLastError** returns ERROR_ NO_MORE_ITEMS.

See Also

SetupDiBuildDriverInfoList

SetupDiGetActualSectionToInstall

```
BOOLEAN
  SetupDiGetActualSectionToInstall(
    IN HINF  InfHandle,
    IN PCTSTR  InfSectionName,
    OUT PTSTR  InfSectionWithExt,  OPTIONAL
    IN DWORD  InfSectionWithExtSize,
    OUT PDWORD  RequiredSize,  OPTIONAL
    OUT PTSTR  *Extension  OPTIONAL
    );
```

SetupDiGetActualSectionToInstall finds the appropriate *DDInstall* section to use when installing a device from a device INF file.

Parameters

InfHandle

Supplies the handle of the INF file that contains the *DDInstall* section.

InfSectionName

Supplies a pointer to the name of the *DDInstall* section as specified by the driver node being installed.

InfSectionWithExt

Supplies a pointer to a character buffer to receive the name of the *DDInstall* section that should be used for installation. If this parameter is NULL, *InfSectionWithExtSize* must be 0. The caller can use the NULL value to determine the required buffer size, because the function returns TRUE and *RequiredSize* is set to the size, in characters, necessary to store the *DDInstall* section name.

InfSectionWithExtSize

Supplies the size, in characters, of the *InfSectionWithExt* buffer.

RequiredSize

Supplies a pointer to a variable that receives the size, in characters, required to store the actual *DDInstall* section name, including the terminating NULL.

Extension

Supplies a pointer to a variable that receives a pointer to the extension, including ".". This parameter is NULL if an extension is not required. *Extension* points to the extension within the caller-supplied buffer. If the *InfSectionWithExt* buffer is not supplied, this variable is not filled in.

Return Value

If the function is successful, it returns TRUE. If the function fails, it returns INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**.

Comments

This function supports an OS/architecture-specific extension that can be used to specify multiple installation behaviors for a single device, dependent on the operating environment. An extension is appended to the INF file *DDInstall* section name to identify that it contains OS/architecture-specific installation instructions. **SetupDiGetActualSectionToInstall** searches for the different *DDInstall* section names in the manner described next.

Starting with the *DDInstall* section name as specified in the driver node (for example, **InstallSec**) the function attempts to find one of the following section names (the search is carried out in the order listed):

In Windows 2000:

1. **InstallSec.NT***platform*
2. **InstallSec.NT**
3. **InstallSec**

In Windows 98:

1. **InstallSec.Win**
2. **InstallSec**

The first *DDInstall* section located is used for the installation. This section name is also used as the base for **Hardware** and **Services** section names. For example, if the *DDInstall* section name found is **InstallSec.NTX86**, the **Services** section name must be named **InstallSec.-NTX86.Services**.

The original *DDInstall* section name specified in the driver node is written to the driver's registry key's **InfSection** value entry. The extension that was found is stored in the key as the REG_SZ value **InfSectionExt**. For example:

```
InfSection      : REG_SZ :    "InstallSec"
InfSectionExt   : REG_SZ :    ".NTX86"
```

If a driver is not selected for the specified device information element, a NULL driver is installed. Upon return, the Install Parameters flags in the device's SP_DEVINSTALL_PARAMS structure indicate whether the system should be restarted or rebooted for the device to start.

The function uses information on the local machine when selecting the correct section name decoration.

See Also

INF DDInstall Section, **SetupDiInstallDevice**

SetupDiGetClassBitmapIndex

```
BOOLEAN
  SetupDiGetClassBitmapIndex(
    IN LPGUID  ClassGuid,  OPTIONAL
    OUT PINT  MiniIconIndex
    );
```

SetupDiGetClassBitmapIndex retrieves the index of the mini-icon supplied for the specified class.

Parameters

ClassGuid

Points to the GUID of the class for which to retrieve the mini-icon.

MiniIconIndex

A pointer to a buffer to receive the index of the mini-icon for the specified class. This buffer is always filled in. It receives the index of the Unknown mini-icon if there is no mini-icon for the specified class.

Return Value

If there is a min-icon for the specified class, the function returns TRUE.

If there is no mini-icon for the specified class, the function returns FALSE and the *Mini-IconIndex* buffer receives the index for the Unknown mini-icon.

See Also

SetupDiDrawMiniIcon, **SetupDiLoadClassIcon**

SetupDiGetClassDescription

```
BOOLEAN
  SetupDiGetClassDescription(
    IN LPGUID  ClassGuid,
    OUT PTSTR  ClassDescription,
    IN DWORD  ClassDescriptionSize,
    OUT PDWORD  RequiredSize  OPTIONAL
    );
```

SetupDiGetClassDescription retrieves the class description associated with the specified setup class GUID.

Parameters

ClassGuid

Supplies the GUID of the setup class whose description is to be retrieved.

ClassDescription

Supplies a pointer to a character buffer that receives the class description.

ClassDescriptionSize

Supplies the size, in characters, of the *ClassDescription* buffer.

RequiredSize

Receives the size, in characters, required to store the class description (including terminating NULL). *RequiredSize* is always less than LINE_LEN.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

Call **SetupDiGetClassDescriptionEx** to retrieve the description of a setup class installed on a remote machine.

See Also

SetupDiBuildClassInfoList, **SetupDiGetClassDescriptionEx**, **SetupDiGetINFClass**

SetupDiGetClassDescriptionEx

```
BOOLEAN
  SetupDiGetClassDescriptionEx(
    IN LPGUID  ClassGuid,
    OUT PTSTR  ClassDescription,
    IN DWORD  ClassDescriptionSize,
    OUT PDWORD  RequiredSize,  OPTIONAL
    IN PCTSTR  MachineName,  OPTIONAL
    IN PVOID  Reserved
    );
```

SetupDiGetClassDescriptionEx retrieves the description of a setup class installed on a local or remote machine.

Parameters

ClassGuid

Supplies the GUID of the setup class whose description is to be retrieved.

ClassDescription

Supplies a pointer to a character buffer that receives the class description.

ClassDescriptionSize

Supplies the size, in characters, of the *ClassDescription* buffer.

RequiredSize

Receives the size, in characters, required to store the class description (including terminating NULL). *RequiredSize* is always less than LINE_LEN.

MachineName

Optionally supplies the name of a remote machine on which the setup class resides. A value of NULL for *MachineName* specifies that the class is installed on the local machine.

Reserved

Must be NULL.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

If there is a friendly name in the registry key for the class, this routine returns the friendly name. Otherwise, this routine returns the class name.

See Also

SetupDiBuildClassInfoList, **SetupDiBuildClassInfoListEx**, **SetupDiGetDeviceInfoList-Detail**, **SetupDiGetINFClass**

SetupDiGetClassDevs

```
HDEVINFO
  SetupDiGetClassDevs(
    IN LPGUID  ClassGuid,  OPTIONAL
    IN PCTSTR  Enumerator,  OPTIONAL
    IN HWND  hwndParent,  OPTIONAL
    IN DWORD  Flags
    );
```

SetupDiGetClassDevs returns a device information set that contains all devices of a specified class.

Parameters

ClassGuid

Optionally points to a class GUID for a setup class or an interface class. If the DIGCF_DEVICEINTERFACE flag is set, *ClassGuid* represents an interface class; otherwise, *ClassGuid* represents a setup class.

If the DIGCF_ALLCLASSES flag is set, this parameter is ignored and the resulting list contains devices of all installed classes.

Enumerator

Optionally points to a string that filters the devices that are returned.

If the DIGCF_DEVICEINTERFACE flag is set, this parameter optionally points to a string representing the PnP name of a particular device. This function only examines this particular device to determine whether it exposes any interfaces of the requested interface class.

If the DIGCF_DEVICEINTERFACE flag is not set, this parameter optionally specifies the name of the PnP enumerator that enumerates the devices of interest. (The names of system-supplied enumerators appear in *register.h.*) This function only examines device instances of this enumerator. If this parameter is NULL, this function retrieves device information for all device instances on the system.

hwndParent

Supplies the handle of the top-level window to be used for any user interface relating to the members of this set.

Flags

Supplies control options used in building the device information set. Can be a combination of the following values:

DIGCF_ALLCLASSES

Return a list of installed devices for all classes. If this flag is set, the *ClassGuid* parameter is ignored.

DIGCF_DEVICEINTERFACE

Return devices that expose interfaces of the interface class specified by *ClassGuid.* If this flag is not set, *ClassGuid* specifies a setup class.

DIGCF_PRESENT

Return only devices that are currently present.

DIGCF_PROFILE

Return only devices that are a part of the current hardware profile.

Return Value

SetupDiGetClassDevs returns a handle to a device information set containing all installed devices matching the specified parameters. If the function fails, it returns INVALID_ HANDLE_VALUE or another appropriate error. To get extended error information, call **GetLastError**.

Comments

The caller of this function must delete the returned device information set when it is no longer needed by calling **SetupDiDestroyDeviceInfoList**.

If DIGCF_DEVICEINTERFACE is set, *ClassGuid* (if used) must point to a device interface class GUID and *Enumerator* (if used) must point to a PnP device name. The returned device information set contains devices that expose interfaces of the requested interface class. Enumerate the interfaces of the devices using **SetupDiEnumDeviceInterfaces**.

If DIGCF_DEVICEINTERFACE is not set, *ClassGuid* (if used) must point to a setup class GUID and *Enumerator* (if used) must specify the name of the PnP enumerator that enumerates the devices of interest.

Call **SetupDiGetClassDevsEx** to retrieve the devices for a class on a remote machine.

See Also

SetupDiCreateDeviceInfoList, **SetupDiDestroyDeviceInfoList**, **SetupDiEnumDevice-Interfaces**, **SetupDiGetClassDevsEx**

SetupDiGetClassDevsEx

```
HDEVINFO
  SetupDiGetClassDevsEx(
    IN LPGUID  ClassGuid,  OPTIONAL
    IN PCTSTR  Enumerator,  OPTIONAL
    IN HWND  hwndParent,  OPTIONAL
    IN DWORD  Flags,
    IN HDEVINFO  DeviceInfoSet,  OPTIONAL
    IN PCTSTR  MachineName,  OPTIONAL
    IN PVOID  Reserved
    );
```

SetupDiGetClassDevsEx returns a device information set that contains all devices of a specified class on a local or remote machine.

Parameters

ClassGuid

Optionally points to a class GUID for a setup class or an interface class. If the DIGCF_ DEVICEINTERFACE flag is set, *ClassGuid* represents an interface class; otherwise, *ClassGuid* represents a setup class.

If the DIGCF_ALLCLASSES flag is set, this parameter is ignored and the resulting list contains devices of all installed classes.

Enumerator

Optionally points to a string that filters the devices that are returned.

If the DIGCF_DEVICEINTERFACE flag is set, this parameter optionally points to a string representing the PnP name of a particular device. This function only examines this particular device to determine whether it exposes any interfaces of the requested interface class.

If the DIGCF_DEVICEINTERFACE flag is not set, this parameter optionally specifies the name of the PnP enumerator that enumerates the devices of interest. (The names of system-supplied enumerators appear in *register.h.*) This function only examines device instances of this enumerator. If this parameter is NULL, this function retrieves device information for all device instances on the system.

hwndParent

Supplies the handle of the top-level window to be used for any user interface relating to the members of this set.

Flags

Supplies control options used in building the device information set. Can be a combination of the following values:

DIGCF_ALLCLASSES

Return a list of installed devices for all classes. If this flag is set, the *ClassGuid* parameter is ignored.

DIGCF_DEVICEINTERFACE

Return devices that expose interfaces of the interface class specified by *ClassGuid.* If this flag is not set, *ClassGuid* specifies a setup class.

DIGCF_PRESENT

Return only devices that are currently present.

DIGCF_PROFILE

Return only devices that are a part of the current hardware profile.

DeviceInfoSet

Optionally supplies the handle of an existing device information set into which this function adds the requested device information.

If *DeviceInfoSet* is nonNULL, then the device information set it specifies is returned on success, with the retrieved device information added. If this parameter is NULL, this function creates a new device information set.

If *DeviceInfoSet* is nonNULL and *ClassGuid* specifies a setup class, then the associated class of *DeviceInfoSet* (if any) must match the *ClassGuid. ClassGuid* specifies a setup class if it is nonNULL and the DIGCF_DEVICEINTERFACE flag is not set.

MachineName

Optionally supplies the name of a remote machine on which the devices reside. A value of NULL for *MachineName* specifies that the class is installed on the local machine.

Reserved

Must be NULL.

Return Value

SetupDiGetClassDevsEx returns a handle to a device information set containing all devices matching the specified parameters. If the function fails, it returns INVALID_HANDLE_VALUE or another appropriate error. To get extended error information, call **GetLast-Error**.

Comments

The caller of this function must delete the returned device information set when it is no longer needed by calling **SetupDiDestroyDeviceInfoList**.

If DIGCF_DEVICEINTERFACE is set, *ClassGuid* (if used) must point to a device interface class GUID and *Enumerator* (if used) must point to a PnP device name. The returned device information set contains devices that expose interfaces of the requested interface class. Enumerate the interfaces of the devices using **SetupDiEnumDeviceInterfaces**.

If DIGCF_DEVICEINTERFACE is not set, *ClassGuid* (if used) must point to a setup class GUID and *Enumerator* (if used) must specify the name of the PnP enumerator that enumerates the devices of interest.

A driver can use this function to get a list of device interfaces of a particular class that are exposed by devices of a particular setup class. For example, to get a list of the device interfaces of interface class "mounted device" that are exposed by devices in the setup class "Volume":

1. Create a device information set (**SetupDiCreateDeviceInfoList[Ex]**) with an associated setup class of "Volume".
2. Call **SetupDiGetClassDevsEx**, specifying:
   - *ClassGuid* with the GUID for the interface class "mounted device".
   - *Flags* with DIGCF_DEVICEINTERFACE set.

- *DeviceInfoSet* with the HDEVINFO returned in step (1). The associated setup class of this HDEVINFO is "Volume".

In this kind of call to **SetupDiGetClassDevsEx**, the device interfaces retrieved are filtered based on whether their corresponding device's setup class matches that of the device information set.

See Also

SetupDiCreateDeviceInfoListEx, **SetupDiDestroyDeviceInfoList**, **SetupDiEnumDeviceInterfaces**

SetupDiGetClassImageIndex

```
BOOLEAN
  SetupDiGetClassImageIndex(
    IN PSP_CLASSIMAGELIST_DATA  ClassImageListData,
    IN LPGUID  ClassGuid,
    OUT PINT  ImageIndex
    );
```

SetupDiGetClassImageIndex retrieves the index within the class image list of a specified class.

Parameters

ClassImageListData

Supplies a pointer to an SP_CLASSIMAGELIST_DATA structure that contains the class's image.

ClassGuid

Supplies a pointer to the GUID for the class.

ImageIndex

Supplies a pointer to a variable that receives the index of the specified class's image within the class image list.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

See Also

SetupDiGetClassImageList, **SetupDiGetClassImageListEx**

SetupDiGetClassImageList

```
BOOLEAN
  SetupDiGetClassImageList(
    OUT PSP_CLASSIMAGELIST_DATA  ClassImageListData
    );
```

SetupDiGetClassImageList builds an image list that contains bitmaps for every installed class and returns the list in a data structure.

Parameters

ClassImageListData

Supplies a pointer to an SP_CLASSIMAGELIST_DATA structure to receive information regarding the class image list, including a handle to the image list. The **cbSize** field of this structure must be initialized with the size of the structure, in bytes, before calling this function or it will fail.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

Comments

The image list built by this function should be destroyed by calling **SetupDiDestroyClass-ImageList**.

Call **SetupDiGetClassImageListEx** to retrieve the image list for classes installed on a remote machine.

See Also

SetupDiDestroyClassImageList, **SetupDiGetClassImageListEx**

SetupDiGetClassImageListEx

```
BOOLEAN
  SetupDiGetClassImageListEx(
    OUT PSP_CLASSIMAGELIST_DATA  ClassImageListData,
    IN PCSTR  MachineName,  OPTIONAL
    IN PVOID  Reserved
    );
```

SetupDiGetClassImageListEx builds an image list of bitmaps for every class installed on a local or remote machine.

Parameters

ClassImageListData

Supplies a pointer to an SP_CLASSIMAGELIST_DATA structure to receive information regarding the class image list, including a handle to the image list. The **cbSize** field of this structure must be initialized with the size of the structure, in bytes, before calling this function or it will fail.

MachineName

Optionally supplies the name of a remote machine for whose classes the bitmap list is to be built. If *MachineName* is NULL, this function builds the list for the local machine.

Reserved

Must be NULL.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

Comments

The image list built by this function should be destroyed by calling **SetupDiDestroy-ClassImageList**.

Note Class-specific icons on a remote machine can only be displayed if the class is also present on the local machine. Thus, if the remote machine has class *X*, but class *X* is not installed locally, then the generic (unknown) icon will be returned.

See Also

SetupDiDestroyClassImageList, **SetupDiGetClassImageList**

SetupDiGetClassInstallParams

```
BOOLEAN
  SetupDiGetClassInstallParams(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    OUT PSP_CLASSINSTALL_HEADER  ClassInstallParams,  OPTIONAL
    IN DWORD  ClassInstallParamsSize,
    OUT PDWORD  RequiredSize  OPTIONAL
    );
```

SetupDiGetClassInstallParams retrieves class install parameters for a device information set or a particular device information element.

Parameters

DeviceInfoSet

Supplies a handle to the device information set that contains the class install parameters to retrieve.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that contains the class install parameters to retrieve. If this parameter is not specified, the class install parameters retrieved are associated with the device information set for the global class driver list.

ClassInstallParams

Supplies a pointer to a buffer that contains a class install header structure. This structure must have its **cbSize** field set to **sizeof**(SP_CLASSINSTALL_HEADER) on input or the buffer is considered to be invalid. On output, the **InstallFunction** field is filled with the DI_FUNCTION code for the class install parameters being retrieved. If the buffer is large enough, it receives the class install parameters structure specific to the function code. If *ClassInstallParams* is not specified, *ClassInstallParamsSize* must be 0.

ClassInstallParamsSize

Supplies the size, in bytes, of the *ClassInstallParams* buffer. If the buffer is supplied, it must be at least as large as **sizeof**(SP_CLASSINSTALL_HEADER). If the buffer is not supplied, *ClassInstallParamsSize* must be 0.

RequiredSize

Supplies a pointer to a variable to receive the number of bytes required to store the class install parameters.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

The class install parameters are specific to a particular device installation request (DI_FUNCTION) that is stored in the **ClassInstallHeader** field located at the beginning of the *ClassInstallParams* buffer.

See Also

SetupDiSetClassInstallParams

SetupDiGetDeviceInfoListClass

```
BOOLEAN
  SetupDiGetDeviceInfoListClass(
    IN HDEVINFO  DeviceInfoSet,
    OUT LPGUID  ClassGuid
    );
```

SetupDiGetDeviceInfoListClass retrieves the class GUID associated with a device information set if it has an associated class.

Parameters

DeviceInfoSet

Supplies a handle to the device information set to query.

ClassGuid

Supplies a pointer to the variable that receives the GUID for the associated class.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

If the specified device information set does not have an associated class because a class GUID was not specified when the set was created with **SetupDiCreateDeviceInfoList**, the function fails. In this case, a call to **GetLastError** returns ERROR_NO_ASSOCIATED_ CLASS.

If a device information set is for a remote machine, use **SetupDiGetDeviceInfoListDetail** to get the associated remote machine handle and machine name.

See Also

SetupDiCreateDeviceInfoList, **SetupDiGetClassDevs**, **SetupDiGetDeviceInfoListDetail**

SetupDiGetDeviceInfoListDetail

```
BOOLEAN
  SetupDiGetDeviceInfoListDetail(
    IN HDEVINFO  DeviceInfoSet,
    OUT PSP_DEVINFO_LIST_DETAIL_DATA  DeviceInfoSetDetailData
    );
```

SetupDiGetDeviceInfoListDetail retrieves information associated with a device information set including the class GUID, remote machine handle, and remote machine name.

Parameters

DeviceInfoSet

Supplies a handle to the device information set to query.

DeviceInfoSetDetailData

Supplies a pointer to a caller-allocated SP_DEVINFO_LIST_DETAIL_DATA structure that receives the device information set information.

The caller must set **cbSize** to **sizeof**(SP_DEVINFO_LIST_DETAIL_DATA) or the function will fail with **GetLastError** returning ERROR_INVALID_USER_BUFFER.

If the function completes successfully, **ClassGuid** contains the class GUID associated with the device information set or GUID_NULL.

If the function completes successfully and the device information set is for a remote machine, **RemoteMachineHandle** contains the ConfigMgr32 machine handle for accessing the remote machine and **RemoteMachineName** contains the name of the remote machine. If there is a remote handle for the device information set, it must be used when calling **CM_**_Xxx_**_Ex** functions because the DevInst handles are relative to the remote handle.

If the device information set is for the local machine, **RemoteMachineHandle** is NULL and **RemoteMachineName** is an empty string.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

If the parameters are valid, **SetupDiGetDeviceInfoListDetail** sets values in the *DeviceInfo-SetDetailData* structure (except for the **cbSize** field) and returns status NO_ERROR.

See Also

SetupDiCreateDeviceInfoListEx, **SetupDiGetClassDevsEx**, **SetupDiGetDevice-InfoListClass**

SetupDiGetDeviceInstallParams

```
BOOLEAN
  SetupDiGetDeviceInstallParams(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    OUT PSP_DEVINSTALL_PARAMS  DeviceInstallParams
    );
```

SetupDiGetDeviceInstallParams retrieves device install parameters for a device information set or a particular device information element.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing information about the device instance to retrieve.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that contains the device install parameters to retrieve. If this parameter is not specified, the install parameters retrieved are those associated with the device information set.

DeviceInstallParams

Supplies a pointer to an SP_DEVINSTALL_PARAMS structure that receives the device install parameters. The **cbSize** field of this structure must be set to the size, in bytes, of the structure before calling this function.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

See Also

SetupDiSetDeviceInstallParams

SetupDiGetDeviceInstanceId

```
BOOLEAN
  SetupDiGetDeviceInstanceId(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,
    OUT PTSTR  DeviceInstanceId,
    IN DWORD  DeviceInstanceIdSize,
    OUT PDWORD  RequiredSize  OPTIONAL
    );
```

SetupDiGetDeviceInstanceId retrieves the device instance ID associated with a device information element.

Parameters

DeviceInfoSet

Supplies a handle to the device information set that contains the device information element to retrieve.

DeviceInfoData

Supplies a pointer to the SP_DEVINFO_DATA structure for the device information element whose ID is to be retrieved.

DeviceInstanceId

Supplies a pointer to the character buffer that will receive the ID for the specified device information element.

DeviceInstanceIdSize

Supplies the size, in characters, of the *DeviceInstanceId* buffer.

RequiredSize

Supplies a pointer to the variable that receives the number of characters required to store the device instance ID.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

See Also

SetupDiCreateDeviceInfo, **SetupDiCreateDevRegKey**, **SetupDiOpenDeviceInfo**, **SetupDiOpenDevRegKey**

SetupDiGetDeviceInterfaceAlias

```
BOOLEAN
  SetupDiGetDeviceInterfaceAlias(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
    IN LPGUID  AliasInterfaceClassGuid,
    OUT PSP_DEVICE_INTERFACE_DATA  AliasDeviceInterfaceData
    );
```

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

Parameters

DeviceInfoSet

Points to the device information set containing the device interface for which to retrieve an alias. This handle is typically returned by **SetupDiGetClassDevs**.

DeviceInterfaceData

Points to a structure that identifies the device interface within the device information set. This pointer is typically returned by **SetupDiEnumDeviceInterfaces**.

AliasInterfaceClassGuid

Is a class GUID specifying the interface class of the alias to retrieve.

AliasDeviceInterfaceData

Points to a caller-allocated buffer that contains, on successful return, a completed SP_DEVICE_INTERFACE_DATA structure that identifies the requested alias. The caller must set *AliasDeviceInterfaceData*.**cbSize** to **sizeof**(SP_DEVICE_INTERFACE_DATA) before calling this function.

Return Value

SetupDiGetDeviceInterfaceAlias returns TRUE if the function completed without error. If the function completed with an error, FALSE is returned and the error code for the failure can be retrieved by calling **GetLastError**.

Possible errors returned by **GetLastError** include:

Error	Meaning
ERROR_INVALID_PARAMETER	Invalid *DeviceInfoSet* or invalid *DeviceInterfaceData* parameter.
ERROR_NO_SUCH_INTERFACE_DEVICE	There is no alias of class *AliasInterfaceClassGuid* for the specified device interface.
ERROR_INVALID_USER_BUFFER	Invalid *AliasDeviceInterfaceData* buffer.

Comments

SetupDiGetDeviceInterfaceAlias can be used to locate a device that exposes more than one interface. For example, consider a disk that can be part of a fault-tolerant volume and can contain encrypted data. The function driver for the disk device could register a fault-tolerant-volume interface and an encrypted-volume interface. These interfaces are device interface aliases if the function driver registers them with identical reference strings and they refer to the same device. (The reference strings will likely be NULL and therefore are equal.) To locate such a multi-interface device, first locate all available devices that expose one of the interfaces, such as the fault-tolerant-volume interface, using **SetupDiGetClass-Devs** and **SetupDiEnumDeviceInterfaces**. Then, pass a device with the first interface (fault-tolerant-volume) to **SetupDiGetDeviceInterfaceAlias** and request an alias of the other interface class (encrypted-volume).

If the requested alias exists but the caller-supplied *AliasDeviceInterfaceData* buffer is invalid, this function successfully adds the device interface element to *DevInfoSet* but returns FALSE for the return value. In this case, **GetLastError** returns ERROR_ INVALID_USER_BUFFER.

See Also

SetupDiEnumDeviceInterfaces, **SetupDiGetClassDevs**

SetupDiGetDeviceInterfaceDetail

```
BOOLEAN
  SetupDiGetDeviceInterfaceDetail(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
    OUT PSP_DEVICE_INTERFACE_DETAIL_DATA  DeviceInterfaceDetailData,  OPTIONAL

IN DWORD  DeviceInterfaceDetailDataSize,
    OUT PDWORD  RequiredSize,  OPTIONAL
    OUT PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
    );
```

SetupDiGetDeviceInterfaceDetail returns details about a particular device interface.

Parameters

DeviceInfoSet

Points to the device information set containing the interface and its underlying device. This handle is typically returned by **SetupDiGetClassDevs**.

DeviceInterfaceData

Points to a structure that identifies the interface, typically returned by **SetupDiEnum-DeviceInterfaces**.

DeviceInterfaceDetailData

Optionally points to a caller-allocated buffer to receive information about the specified interface. The caller must set *DeviceInterfaceDetailData*.**cbSize** to **sizeof**(SP_DEVICE_INTERFACE_DETAIL_DATA) before calling this function. The **cbSize** field always contains the size of the fixed part of the data structure, not a size reflecting the variable-length string at the end.

This parameter must be NULL if *DeviceInterfaceDetailSize* is zero.

DeviceInterfaceDetailDataSize

Specifies the size of the *DeviceInterfaceDetailData* buffer. The buffer must be at least (**offsetof**(SP_DEVICE_INTERFACE_DETAIL_DATA, **DevicePath**) + **sizeof**(TCHAR)) bytes, to contain the fixed part of the structure and a single NULL to terminate an empty MULTI_SZ string.

This parameter must be zero if *DeviceInterfaceDetailData* is NULL.

RequiredSize

Optionally points to a caller-allocated variable to receive the required size of the *Device-InterfaceDetailData* buffer. This size includes the size of the fixed part of the structure plus the number of bytes required for the variable-length device path string.

DeviceInfoData

Optionally points to a caller-allocated buffer to receive information about the device that exposes the requested interface. The caller must set *DeviceInfoData*.**cbSize** to **sizeof**(SP_DEVINFO_DATA).

Return Value

SetupDiGetDeviceInterfaceDetail returns TRUE if the function completed without error. If the function completed with an error, FALSE is returned and the error code for the failure can be retrieved by calling **GetLastError**.

Comments

Using this function to get details about an interface is typically a two-step process:

1. Get the required buffer size. Call **SetupDiGetDeviceInterfaceDetail** with a NULL *DeviceInterfaceDetailData* pointer, an *DeviceInterfaceDetailDataSize* of zero, and a valid *RequiredSize* variable. In response to such a call, this function returns the required buffer size at *RequiredSize* and fails with **GetLastError** returning ERROR_INSUFFICIENT_BUFFER.
2. Allocate an appropriately sized buffer and call the function again to get the interface details.

The interface detail returned by this function consists of a device path that can be passed to Win32 functions such as **CreateFile**. Do not attempt to parse the device path symbolic name. The device path can be reused across system boots.

SetupDiGetDeviceInterfaceDetail can be used to get just the *DeviceInfoData*. If the interface exists but *DeviceInterfaceDetailData* is NULL, this function fails, **GetLastError** returns ERROR_INSUFFICIENT_BUFFER, and the *DeviceInfoData* structure is filled with information about the device that exposes the interface.

See Also

CreateFile, **SetupDiEnumDeviceInterfaces**, **SetupDiGetClassDevs**

SetupDiGetDeviceRegistryProperty

```
BOOLEAN
  SetupDiGetDeviceRegistryProperty
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,
    IN DWORD  Property,
    OUT PDWORD  PropertyRegDataType,  OPTIONAL
    OUT PBYTE  PropertyBuffer,
    IN DWORD  PropertyBufferSize,
    OUT PDWORD  RequiredSize  OPTIONAL
    );
```

SetupDiGetDeviceRegistryProperty retrieves the specified Plug and Play device property.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing information about the device instance for which to retrieve a Plug and Play property.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure indicating the device instance for which to retrieve the Plug and Play property.

Property

Supplies an ordinal specifying the property to be retrieved. Can be one of the following values:

Code	Property
SPDRP_ADDRESS	Device Address
SPDRP_BUSNUMBER	BusNumber
SPDRP_BUSTYPEGUID	BusTypeGUID
SPDRP_CAPABILITIES	Capabilities
SPDRP_CHARACTERISTICS	Device Characteristics
SPDRP_CLASS	Class
SPDRP_CLASSGUID	ClassGUID
SPDRP_COMPATIBLEIDS	CompatibleIDs
SPDRP_CONFIGFLAGS	ConfigFlags
SPDRP_DEVICEDESC	DeviceDesc
SPDRP_DEVTYPE	Device Type
SPDRP_DRIVER	Driver
SPDRP_ENUMERATOR_NAME	Enumerator Name
SPDRP_EXCLUSIVE	Exclusive access
SPDRP_FRIENDLYNAME	FriendlyName
SPDRP_HARDWAREID	HardwareID
SPDRP_LEGACYBUSTYPE	LegacyBusType
SPDRP_LOCATION_INFORMATION	LocationInformation
SPDRP_LOWERFILTERS	LowerFilters
SPDRP_MFG	Mfg
SPDRP_PHYSICAL_DEVICE_OBJECT_NAME	PhysicalDeviceObjectName
SPDRP_SECURITY	Security (binary form)
SPDRP_SERVICE	Service
SPDRP_UI_NUMBER	UiNumber
SPDRP_UI_NUMBER_DESC_FORMAT	Format-message-style string to format UI number
SPDRP_UPPERFILTERS	UpperFilters

PropertyRegDataType

Supplies a pointer to a variable to receive the data type of the property being retrieved. This is one of the standard registry data types.

PropertyBuffer

Supplies a pointer to a buffer to receive the property being retrieved.

PropertyBufferSize

Supplies the length, in bytes, of *PropertyBuffer*.

RequiredSize

Supplies a pointer to a variable to receive the number of bytes required to store the requested property in *PropertyBuffer*.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

See Also

SetupDiSetDeviceRegistryProperty

SetupDiGetDriverInfoDetail

```
BOOLEAN
  SetupDiGetDriverInfoDetail(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN PSP_DRVINFO_DATA  DriverInfoData,
    OUT PSP_DRVINFO_DETAIL_DATA  DriverInfoDetailData,  OPTIONAL
    IN DWORD  DriverInfoDetailDataSize,
    OUT PDWORD  RequiredSize  OPTIONAL
    );
```

SetupDiGetDriverInfoDetail retrieves detailed information for a specified driver information element.

Parameters

DeviceInfoSet

Supplies a handle to the device information set that contains a driver information structure for which to retrieve details.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that contains a driver information structure for which to retrieve details. If this parameter is not specified, the driver referenced is a member of the global class driver list owned by the device information set.

DriverInfoData

Supplies a pointer to an SP_DRVINFO_DATA structure that specifies the driver for which details are to be retrieved.

DriverInfoDetailData

Supplies a pointer to an SP_DRVINFO_DETAIL_DATA structure that receives detailed information about the specified driver. If this parameter is not specified, *DriverInfoDetail-DataSize* must be 0. If this parameter is specified, the **cbSize** field of this structure must be set to the size, in bytes, of the structure before calling **SetupDiGetDriverInfoDetail**.

DriverInfoDetailDataSize

Supplies the size, in bytes, of the *DriverInfoDetailData* buffer.

RequiredSize

Supplies a pointer to a variable that receives the number of bytes required to store the detailed driver information. This value includes both the size of the structure and the additional bytes required for the variable-length character buffer at the end that holds the hardware ID and the compatible IDs MULTI_SZ list.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

If the specified driver information member and the caller-supplied buffer are both valid, this function is guaranteed to fill in all static fields in the SP_DRVINFO_DETAIL_DATA structure and as many IDs as possible in the variable-length buffer at the end while still maintaining MULTI_SZ format. In this case, the function returns FALSE and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER. If specified, *RequiredSize* contains the total number of bytes required for the structure with all IDs.

See Also

SetupDiEnumDriverInfo, **SetupDiGetSelectedDriver**

SetupDiGetDriverInstallParams

```
BOOLEAN
  SetupDiGetDriverInstallParams(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN PSP_DRVINFO_DATA  DriverInfoData,
    OUT PSP_DRVINSTALL_PARAMS  DriverInstallParams
    );
```

SetupDiGetDriverInstallParams retrieves install parameters for the specified driver.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing a driver information structure for which to retrieve installation parameters.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that contains a driver information structure for which to retrieve installation parameters. If *DeviceInfoData* is not specified, the driver referenced is a member of the global class driver list owned by the device information set.

DriverInfoData

Supplies a pointer to an SP_DRVINFO_DATA structure that specifies the driver for which install parameters are to be retrieved.

DriverInstallParams

Supplies a pointer to an SP_DRVINSTALL_PARAMS structure to receive the install parameters for this driver. The **cbSize** field of this structure must be set to the size, in bytes, of the structure before calling this function.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

See Also

SetupDiSetDriverInstallParams

SetupDiGetHwProfileFriendlyName

```
BOOLEAN
  SetupDiGetHwProfileFriendlyName(
    IN DWORD  HwProfile,
    OUT PTSTR  FriendlyName,
    IN DWORD  FriendlyNameSize,
    OUT PDWORD  RequiredSize  OPTIONAL
    );
```

SetupDiGetHwProfileFriendlyName retrieves the friendly name associated with a hardware profile ID.

Parameters

HwProfile

Supplies the hardware profile ID associated with the friendly name to retrieve. If this parameter is 0, the friendly name for the current hardware profile is retrieved.

FriendlyName

Supplies a pointer to a character buffer to receive the friendly name.

FriendlyNameSize

Supplies the size, in characters, of the *FriendlyName* buffer.

RequiredSize

Supplies a pointer to a variable to receive the number of characters required to store the friendly name (including the terminating NULL).

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

Call **SetupDiGetHwProfileFriendlyNameEx** to get the friendly name of a hardware profile ID on a remote machine.

See Also

SetupDiGetHwProfileFriendlyNameEx, **SetupDiGetHwProfileList**

SetupDiGetHwProfileFriendlyNameEx

```
BOOLEAN
  SetupDiGetHwProfileFriendlyNameEx(
    IN DWORD  HwProfile,
    OUT PTSTR  FriendlyName,
    IN DWORD  FriendlyNameSize,
    OUT PDWORD  RequiredSize,  OPTIONAL
    IN PCTSTR  MachineName,  OPTIONAL
    IN PVOID  Reserved
    );
```

SetupDiGetHwProfileFriendlyNameEx retrieves the friendly name associated with a hardware profile ID on a local or remote machine.

Parameters

HwProfile

Supplies the hardware profile ID associated with the friendly name to retrieve. If this parameter is 0, the friendly name for the current hardware profile is retrieved.

FriendlyName

Supplies a pointer to a character buffer to receive the friendly name.

FriendlyNameSize

Supplies the size, in characters, of the *FriendlyName* buffer.

RequiredSize

Supplies a pointer to a variable to receive the number of characters required to store the friendly name (including the terminating NULL).

MachineName

Optionally supplies the name of a remote machine on which the hardware profile ID resides. If *MachineName* is NULL, the hardware profile ID is on the local machine.

Reserved

Must be NULL.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

See Also

SetupDiGetHwProfileFriendlyName, **SetupDiGetHwProfileListEx**

SetupDiGetHwProfileList

```
BOOLEAN
  SetupDiGetHwProfileList
    OUT PDWORD  HwProfileList,
    IN DWORD  HwProfileListSize,
    OUT PDWORD  RequiredSize,
    OUT PDWORD  CurrentlyActiveIndex  OPTIONAL
    );
```

SetupDiGetHwProfileList retrieves a list of all currently defined hardware profile IDs.

Parameters

HwProfileList

Supplies a pointer to an array to receive the list of currently defined hardware profile IDs.

HwProfileListSize

Supplies the number of DWORDs in the *HwProfileList* buffer.

RequiredSize

Supplies a pointer to a variable that receives the number of hardware profiles currently defined. If the number is larger than *HwProfileListSize*, the list is truncated to fit the array size. The value returned in *RequiredSize* indicates the array size required to store the entire list of hardware profiles. In this case, the function fails and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

CurrentlyActiveIndex

Supplies a pointer to a variable that receives the index within the returned *HwProfileList* of the currently active hardware profile.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

Call **SetupDiGetHwProfileListEx** to retrieve the hardware profile IDs for a remote machine.

See Also

SetupDiCreateDevRegKey, **SetupDiOpenDevRegKey**

SetupDiGetHwProfileListEx

```
BOOLEAN
  SetupDiGetHwProfileListEx
    OUT PDWORD  HwProfileList,
    IN DWORD  HwProfileListSize,
    OUT PDWORD  RequiredSize,
    OUT PDWORD  CurrentlyActiveIndex,  OPTIONAL
    IN PCTSTR  MachineName,  OPTIONAL
    IN PVOID  Reserved
    );
```

SetupDiGetHwProfileListEx retrieves a list of all currently defined hardware profile IDs on a local or remote machine.

Parameters

HwProfileList

Supplies a pointer to an array to receive the list of currently defined hardware profile IDs.

HwProfileListSize

Supplies the number of DWORDs in the *HwProfileList* buffer.

RequiredSize

Supplies a pointer to a variable that receives the number of hardware profiles currently defined. If the number is larger than *HwProfileListSize*, the list is truncated to fit the array size. The value returned in *RequiredSize* indicates the array size required to store the entire list of hardware profiles. In this case, the function fails and a call to **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

CurrentlyActiveIndex

Supplies a pointer to a variable that receives the index within the returned *HwProfileList* of the currently active hardware profile.

MachineName

Optionally supplies the name of a remote machine for which to retrieve the list of hardware profile IDs. If *MachineName* is NULL, the list is retrieved for the local machine.

Reserved

Must be NULL.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

See Also

SetupDiGetHwProfileFriendlyNameEx

SetupDiGetINFClass

```
BOOLEAN
  SetupDiGetINFClass(
    IN PCTSTR  InfName,
    OUT LPGUID  ClassGuid,
    OUT PTSTR  ClassName,
    IN DWORD  ClassNameSize,
    OUT PDWORD  RequiredSize  OPTIONAL
    );
```

SetupDiGetINFClass returns the class of a specified device INF file.

Parameters

InfName

Supplies the name of a device INF file. This name can include a path. However, if just the filename is specified, the file is searched for in each directory listed in the **DevicePath** entry under the **HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion** subkey of the registry.

ClassGuid

Receives the class GUID for the specified INF file. If the INF file does not specify a class name, this variable is set to GUID_NULL. Call **SetupDiClassGuidsFromName** to determine if one or more classes with this name are already installed.

ClassName

Receives the name of the class for the specified INF file. If the INF file does not specify a class name, but does specify a GUID, this buffer receives the name retrieved by calling **SetupDiClassNameFromGuid**. However, if **SetupDiClassNameFromGuid** cannot retrieve a class name (for example, the class is not installed), it returns an empty string.

ClassNameSize

Supplies the size, in characters, of the *ClassName* buffer.

RequiredSize

Receives the number of characters required to store the class name (including terminating NULL). *RequiredSize* is always less than MAX_CLASS_NAME_LEN.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

This function works with device INF files on Windows NT 4.0, Windows 2000, and higher and Windows 9x. Do not use this function with legacy INF files.

See Also

SetupDiBuildClassInfoList, **SetupDiClassGuidsFromName**, **SetupDiClassNameFromGuid**, **SetupDiGetClassDescription**

SetupDiGetSelectedDevice

```
BOOLEAN
  SetupDiGetSelectedDevice(
    IN HDEVINFO  DeviceInfoSet,
    OUT PSP_DEVINFO_DATA  DeviceInfoData
    );
```

SetupDiGetSelectedDevice retrieves the currently-selected device for the specified device information set.

Parameters

DeviceInfoSet

Supplies a handle to the device information set from which to retrieve the selected device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that receives the currently-selected device. The caller must set **cbSize** to **sizeof**(SP_DEVINFO_DATA). If a device is not currently selected, the function fails and a call to **GetLastError** returns ERROR_NO_DEVICE_SELECTED.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

SetupDiGetSelectedDevice is usually used by an installation wizard.

See Also

SetupDiSetSelectedDevice, SP_DEVINFO_DATA

SetupDiGetSelectedDriver

```
BOOLEAN
  SetupDiGetSelectedDriver(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    OUT PSP_DRVINFO_DATA  DriverInfoData
    );
```

SetupDiGetSelectedDriver retrieves the member of a driver list that has been selected as the driver to install.

Parameters

DeviceInfoSet

Supplies a handle to the device information set to query.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that indicates the device instance for which to retrieve the selected driver. If this parameter is NULL, the selected class driver for the global class driver list is retrieved.

DriverInfoData

Supplies a pointer to an SP_DRVINFO_DATA structure that receives information about the selected driver.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**. If a driver has not been selected for the specified device instance, the logged error is ERROR_NO_DRIVER_SELECTED.

See Also

SetupDiSetSelectedDriver

SetupDiGetWizardPage

```
HPROPSHEETPAGE
  SetupDiGetWizardPage(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN PSP_INSTALLWIZARD_DATA  InstallWizardData,
    IN DWORD  PageType,
    IN DWORD  Flags
    );
```

This function is reserved for system use. For information on wizard pages, see the DIF_NEWDEVICEWIZARD_*XXX* requests such as DIF_NEWDEVICEWIZARD_FINISHINSTALL.

SetupDiInstallClass

```
BOOLEAN
  SetupDiInstallClass(
    IN HWND  hwndParent,  OPTIONAL
    IN PCTSTR  InfFileName,
    IN DWORD  Flags,
    IN HSPFILEQ  FileQueue  OPTIONAL
    );
```

SetupDiInstallClass installs the **ClassInstall32** section of the specified INF file.

Parameters

hwndParent

Supplies the handle of the parent window for any user interface used to install this class.

InfFileName

Specifies the name of the INF file containing a **ClassInstall32** section.

Flags

These flags control the installation process. Can be a combination of the following:

DI_NOVCP

Set this flag if *FileQueue* is supplied. DI_NOVCP instructs the **SetupInstallFromInf-Section** function not to create a queue of its own and to use the caller-supplied queue instead. If this flag is set, files are not copied just queued.

DI_NOBROWSE

Set this flag to disable browsing in the event a copy operation cannot find a specified file. If the caller supplies a file queue, this flag is ignored.

DI_FORCECOPY

Set this flag to always copy files, even if they are already present on the user's machine. If the caller supplies a file queue, this flag is ignored.

DI_QUIETINSTALL

Set this flag to suppress the user interface unless absolutely necessary. For example, do not display the progress dialog. If the caller supplies a file queue, this flag is ignored.

FileQueue

If the DI_NOVCP flag is set, this parameter supplies a handle to a file queue where file operations should be queued but not committed.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

This function is called by a class installer when it installs a device of a new device class.

To install an interface class or a device class, use **SetupDiInstallClassEx**.

See Also

SetupDiCallClassInstaller, **SetupDiInstallClassEx**

SetupDiInstallClassEx

```
BOOLEAN
  SetupDiInstallClassEx(
    IN HWND  hwndParent,  OPTIONAL
    IN PCTSTR  InfFileName,  OPTIONAL
    IN DWORD  Flags,
    IN HSPFILEQ  FileQueue,  OPTIONAL
    IN LPGUID  InterfaceClassGuid,  OPTIONAL
    IN PVOID  Reserved1,
    IN PVOID  Reserved2
    );
```

SetupDiInstallClassEx installs a class installer or an interface class.

Parameters

hwndParent

Optionally supplies the handle of the parent window for any user interface used to install this class.

InfFileName

Optionally specifies the name of an INF file.

If this function is being used to install a class installer, the INF file contains a **Class-Install32** section and this parameter must be nonNULL.

If this function is being used to install an interface class, the INF file contains an **InterfaceInstall32** section.

Flags

Specifies flags that control the installation process. Can be a combination of the following:

DI_NOVCP

Set this flag if *FileQueue* is supplied. DI_NOVCP instructs the **SetupInstallFromInf-Section** function to not create a queue of its own and to use the caller-supplied queue instead. If this flag is set, files are not copied just queued.

DI_NOBROWSE

Set this flag to disable browsing in the event a copy operation cannot find a specified file. If the caller supplies a file queue, this flag is ignored.

DI_FORCECOPY

Set this flag to always copy files, even if they are already present on the user's machine. If the caller supplies a file queue, this flag is ignored.

DI_QUIETINSTALL

Set this flag to suppress the user interface unless absolutely necessary. For example, do not display the progress dialog. If the caller supplies a file queue, this flag is ignored.

FileQueue

If the DI_NOVCP flag is set, this parameter supplies a handle to a file queue where file operations should be queued but not committed.

InterfaceClassGuid

Optionally points to a GUID specifying an interface class to be installed. If this parameter is nonNULL, this function is being used to install the interface class represented by the GUID. If this parameter is NULL, this function is being used to install a class installer.

Reserved1

Reserved. Must be zero.

Reserved2

Reserved. Must be zero.

Return Value

SetupDiInstallClassEx returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

SetupDiInstallClassEx is typically called by a class installer to install a new device setup class or a new device interface class. Note that an interface class can also be installed automatically as a result of installing the device interfaces for a device instance (**SetupDiInstall-DeviceInterfaces**).

See Also

SetupDiCallClassInstaller, **SetupDiInstallDeviceInterfaces**

SetupDiInstallDevice

```
BOOLEAN
  SetupDiInstallDevice(
    IN HDEVINFO  DeviceInfoSet,
    IN OUT PSP_DEVINFO_DATA  DeviceInfoData
    );
```

SetupDiInstallDevice is the default handler for the DIF_INSTALLDEVICE installation request. It installs a driver for a device.

Parameters

DeviceInfoSet

Supplies a handle to the device information set for the local machine that contains the device to be installed.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device to be installed. This is an IN OUT parameter because the **DevInst** field of the structure can be updated with a new handle value upon return.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

This function installs a driver from the INF file. SetupAPI's definition of the "driver" is really a "driver node". Therefore, when this function installs a driver, that includes installing the service(s) for the device, the driver files, any device-specific coinstallers, property-page providers, and control-panel applets, and registering any device interfaces. A successful installation includes, but is not limited to, the following steps:

- Create a driver key in the registry and write appropriate entries (such as **InfPath** and **ProviderName**).
- Locate and process the *DDInstall* section for the device. The section might be OS/architecture-specific. The *DDInstall* section's **AddReg** and **DelReg** entries are directed at the device's software key. Locate and process the *DDInstall*.**HW** section whose **AddReg** and **DelReg** entries are directed at the device's hardware key. Locate and process the *DDInstall*.**LogConfigOverride** section, if present, to supply an override LogConfig for the device. Locate and process the *DDInstall*.**Services** section to add services for the device (and potentially remove any old services that are no longer necessary).
- Copy the INF to system INF directory.
- Possibly perform the other file operations, based on flag settings in the device installation parameters.

  If the DI_NOFILECOPY flag and the DI_NOVCP flag are *clear*, perform any specified file in the *DDInstall* section. If the DI_NOVCP flag is set, queue any file operations. If the DI_NOFILECOPY flag is set, do not copy the files; this flag might be set if, for example, a DIF_INSTALLDEVICEFILES operation was already performed for this device installation.
- Load the driver(s) for the device, including the function driver and any upper or lower filter drivers.
- Call the drivers at their AddDevice routines.
- Start the device (send an IRP_MN_START_DEVICE).

Setup does not start the device if the DI_NEEDRESTART, DI_NEEDREBOOT, or DI_DONOTCALLCONFIGMG flag in the SP_DEVINSTALL_PARAMS structure is set.

A class installer should return ERROR_DI_DO_DEFAULT or call this function when handling a DIF_INSTALLDEVICE request. This function performs numerous tasks for device installation and that list of tasks might be expanded in future releases. If a class installer performs device installation without calling this function, the class installer might not work properly on future versions of the operating system.

See Also

DIF_INSTALLDEVICE, **SetupDiCallClassInstaller**, **SetupDiInstallDriverFiles**

SetupDiInstallDeviceInterfaces

```
BOOLEAN
  SetupDiInstallDeviceInterfaces(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData
    );
```

SetupDiInstallDeviceInterfaces is the default handler for the DIF_ INSTALLINTERFACES installation request. It installs the interfaces listed in a *DDInstall*.**Interfaces** section of a device INF file.

Parameters

DeviceInfoSet

Points to the device information set containing the device whose interfaces are to be installed. The device information set must not contain remote elements.

DeviceInfoData

Points to an SP_DEVINFO_DATA structure that identifies a device in the device information set.

Return Value

SetupDiInstallDeviceInterfaces returns TRUE if the function completed without error. If the function completed with an error, FALSE is returned and the error code for the failure can be retrieved by calling **GetLastError**.

Comments

SetupDiInstallDeviceInterfaces processes each **AddInterface** entry in the INF file and creates each interface using **SetupDiCreateDeviceInterface**.

The *DeviceInfoSet* must only contain elements on the local machine.

For information on INF file format, see the chapter on *INF File Sections and Directives*.

See Also

DIF_INSTALLINTERFACES, **SetupDiCreateDeviceInterface**

SetupDiInstallDriverFiles

```
BOOLEAN
  SetupDiInstallDriverFiles(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
    );
```

SetupDiInstallDriverFiles is the default handler for the DIF_INSTALLDEVICEFILES installation request. It is similar to the **SetupDiInstallDevice** function; however, it performs only the file copy commands. The **SetupDiInstallDriverFiles** function does not configure the device.

Parameters

DeviceInfoSet

Supplies a handle to the device information set for which driver files are to be installed. The device information set must not contain remote elements.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure indicating a particular member for which to perform file installation.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

A driver must be selected for the specified device information set or element before this function is called.

This function processes the **CopyFiles**, **Delfiles**, and **Renfiles** entries in the selected INF.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiCallClassInstaller, **SetupDiInstallDevice**

SetupDiLoadClassIcon

```
BOOLEAN
  SetupDiLoadClassIcon(
    IN LPGUID  ClassGuid,
    OUT HICON  *LargeIcon,  OPTIONAL

OUT LPINT  MiniIconIndex  OPTIONAL
    );
```

SetupDiLoadClassIcon loads both the large and mini-icon for the specified class.

Parameters

ClassGuid

Supplies the GUID of the class for which the icon(s) should be loaded.

LargeIcon

Supplies a pointer to a variable to receive a handle for the loaded large icon for the specified class. If this parameter is not specified, the large icon is not loaded.

MiniIconIndex

Supplies a pointer to a variable to receive the index of the mini-icon for the specified class. The mini-icon is stored in the device installer's mini-icon cache.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

Comments

The icons of the class are either predefined and loaded from the device installer's internal cache, or they are loaded directly from the class installer's executable. This function queries the registry value **ICON** in the specified class's section. If the **ICON** value is specified, it indicates which mini-icon to load.

If the **ICON** value is negative, the absolute value represents a predefined icon in the class's registry. See ***SetupDiDrawMiniIcon*** for a list of the predefined mini-icons.

If the **ICON** value is positive, it represents an icon in the class installer's executable that will be extracted. The value 1 is reserved. This function also uses the **INSTALLER32** registry value and then the **ENUMPROPPAGES32** registry value to determine which executable to extract the icon(s) from.

See Also

SetupDiDrawMiniIcon, SetupDiGetClassBitmapIndex

SetupDiMoveDuplicateDevice

```
BOOLEAN
  SetupDiMoveDuplicateDevice(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DestinationDeviceInfoData
    );
```

SetupDiMoveDuplicateDevice is the default handler for the DIF_MOVEDEVICE installation request. This function moves a device to a new location of the **Enum** branch of the registry.

Parameters

DeviceInfoSet

Supplies a handle to the device information set for the device which is to be moved. The device information set must not contain remote elements.

DestinationDeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure for the device instance that is the destination of the move. This device instance must contain class install parameters for DIF_MOVEDEVICE or the function fails with an error of ERROR_NO_CLASSINSTALL_PARAMS.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

This function is typically only used by system components and is not called by vendor device installers.

See Also

SP_MOVEDEV_PARAMS

SetupDiOpenClassRegKey

```
HKEY
  SetupDiOpenClassRegKey(
    IN LPGUID  ClassGuid,  OPTIONAL
    IN REGSAM  samDesired
    );
```

SetupDiOpenClassRegKey opens the setup class registry key or a specific class's subkey.

Parameters

ClassGuid

Optionally supplies the GUID of the setup class whose key is to be opened. If this parameter is NULL, the root of the setup class tree (**HKLM\SYSTEM\CurrentControlSet\Control\Class**) is opened.

samDesired

Specifies the access to the key required by the caller.

Return Value

If the function is successful, it returns a handle to an opened registry key where information pertaining to this setup class can be stored/retrieved. If the function fails, it returns INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**.

Comments

This function does not create a registry key if it does not already exist.

The handle returned from this function must be closed by calling **RegCloseKey**.

To open the interface class registry key or a specific interface class subkey, call **SetupDi-OpenClassRegKeyEx**.

See Also

SetupDiOpenClassRegKeyEx, **SetupDiOpenDevRegKey**

SetupDiOpenClassRegKeyEx

```
HKEY
  SetupDiOpenClassRegKeyEx(
    IN LPGUID  ClassGuid,  OPTIONAL
    IN REGSAM  samDesired,
    IN DWORD  Flags,
    IN PCTSTR  MachineName,  OPTIONAL
    IN PVOID  Reserved
    );
```

SetupDiOpenClassRegKeyEx opens the device setup class registry key, the device interface class registry key, or a specific class's subkey. This function opens the specified key on the local machine or on a remote machine.

Parameters

ClassGuid

Optionally points to the GUID of the class for which the registry key is to be opened. If this parameter is NULL, the root of the class tree (**HKLM\SYSTEM\CurrentControlSet\Control\Class**) is opened.

samDesired

Specifies the access to the key required by the caller.

Flags

Specifies the kind of registry key to be opened. Can be one of the following:

DIOCR_INSTALLER

Open a setup class key. If *ClassGuid* is NULL, open the root key of the class installer branch.

DIOCR_INTERFACE

Open an interface class key. If *ClassGuid* is NULL, open the root key of the interface class branch.

MachineName

Optionally points to a string containing the name of a remote machine on which to open the specified key.

Reserved

Reserved. Must be NULL.

Return Value

SetupDiOpenClassRegKeyEx returns a handle to an opened registry key where information pertaining to this setup class can be stored/retrieved. If the function fails, it returns INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**.

Comments

SetupDiOpenClassRegKeyEx does not create a registry key if it does not already exist.

Callers of this function must close the handle returned from this function by calling **RegCloseKey**.

See Also

SetupDiCreateDeviceInterfaceRegKey, **SetupDiOpenDevRegKey**

SetupDiOpenDeviceInfo

```
BOOLEAN
  SetupDiOpenDeviceInfo(
    IN HDEVINFO  DeviceInfoSet,
    IN PCTSTR  DeviceInstanceId,
    IN HWND  hwndParent,  OPTIONAL
    IN DWORD  OpenFlags,
    OUT PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
    );
```

SetupDiOpenDeviceInfo retrieves information about an existing device instance and adds it to the specified device information set. If a device information element already exists for this device instance, the function returns the existing element.

Parameters

DeviceInfoSet

Supplies a handle to a device information set to which the opened device information element should be added.

DeviceInstanceId

Supplies the ID of the device instance. This is the registry path relative to the **Enum** path of the device instance key. For example, **Root*PNP0500\0000**.

hwndParent

Supplies the window handle of the top-level window to use for any user interface related to installing the device.

OpenFlags

Controls how the device information element is opened. Can be one or more of the following:

DIOD_CANCEL_REMOVE

If this flag is specified and the device had been marked for pending removal, the OS cancels the pending removal.

DIOD_INHERIT_CLASSDRVS

If this flag is specified, the resulting device information element inherits the class driver list, if any, associated with the device information set. In addition, if there is a selected driver for the device information set, that same driver is selected for the new device information element.

If the device information element was already present, its class driver list, if any, is replaced with the inherited list.

DeviceInfoData

Supplies a pointer to a variable that receives a context structure that is initialized for the newly-opened device information element. The caller must set **cbSize** to **sizeof**(SP_DEVINFO_DATA).

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

If this device instance is being added to a set that has an associated class, the device class must be the same or the call will fail. In this case, a call to **GetLastError** returns ERROR_CLASS_MISMATCH.

If the new device information element is successfully opened but the caller-supplied *DeviceInfoData* buffer is invalid, this function returns FALSE. In this case, a call to **GetLastError** returns ERROR_INVALID_USER_BUFFER. However, the device information element is added as a new member of the set anyway.

See Also

SetupDiCreateDeviceInfo, **SetupDiDeleteDeviceInfo**, **SetupDiEnumDeviceInfo**, SP_DEVINFO_DATA

SetupDiOpenDeviceInterface

```
BOOLEAN
  SetupDiOpenDeviceInterface(
    IN HDEVINFO  DeviceInfoSet,
    IN PCTSTR  DevicePath,
    IN DWORD  OpenFlags,
    OUT PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData  OPTIONAL
    );
```

SetupDiOpenDeviceInterface retrieves information about an existing device interface and adds it to the specified device information set. This function creates a device information element for the underlying device if one is not already present in the device information set.

Parameters

DeviceInfoSet

Points to a device information set that contains, or will contain, the device that exposes the interface being opened.

DevicePath

Points to a string containing the name of the device interface to be opened. This name is a Win32 device path typcially received in a PnP notification structure or obtained by a previous call to **SetupDiEnumDeviceInterfaces** and its related functions.

OpenFlags

Reserved. Must be zero.

DeviceInterfaceData

Optionally points to a caller-allocated buffer to receive a completed SP_DEVICE_INTERFACE_DATA structure that identfies the interface. The caller must set *DeviceInterfaceData*.**cbSize** to **sizeof**(SP_DEVICE_INTERFACE_DATA) before calling this function.

Return Value

SetupDiOpenDeviceInterface returns TRUE if the function completed without error. If the function completed with an error, it returns FALSE and the error code for the failure can be retrieved by calling **GetLastError**.

Comments

If a device interface element for the interface already exists in *DeviceInfoSet*, **SetupDiOpenDeviceInterface** updates the flags. This function, therefore, can be used to refresh the flags for a device interface. For example, an interface might have been inactive when it was first opened, but has subsequently become active.

If the new device interface is successfully opened, but the caller-supplied *DeviceInterfaceData* buffer is invalid, this function returns FALSE and **GetLastError** returns ERROR_INVALID_USER_BUFFER. The caller's buffer error does not prevent the interface from being opened.

See Also

SetupDiEnumDeviceInterfaces

SetupDiOpenDeviceInterfaceRegKey

```
HKEY
  SetupDiOpenDeviceInterfaceRegKey(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData,
    IN DWORD  Reserved,
    IN REGSAM  samDesired
    );
```

SetupDiOpenDeviceInterfaceRegKey opens the registry subkey that is used by applications and drivers to store information specific to a device interface instance and returns a handle to the key.

Parameters

DeviceInfoSet

Points to a device information set containing the interface and its underlying device.

DeviceInterfaceData

Points to a structure that identifies the device interface, possibly returned by **SetupDi-CreateDeviceInterface** or **SetupDiEnumDeviceInterfaces**.

Reserved

Reserved. Must be zero.

samDesired

Specifies the access to the registry key requested by the caller.

Return Value

SetupDiOpenDeviceInterfaceRegKey returns a handle to the opened registry key. If the function fails, it returns INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**.

Comments

Close the handle returned from by function by calling **RegCloseKey**.

See Also

SetupDiCreateDeviceInterface, **SetupDiCreateDeviceInterfaceRegKey**, **SetupDiEnum-DeviceInterfaces**

SetupDiOpenDevRegKey

```
HKEY
  SetupDiOpenDevRegKey(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,
    IN DWORD  Scope,
    IN DWORD  HwProfile,
    IN DWORD  KeyType,
    IN REGSAM  samDesired
    );
```

SetupDiOpenDevRegKey opens a registry storage key for device-specific configuration information and returns a handle to the key.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing information about the device instance whose registry configuration storage key is to be opened.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure indicating the device instance for which to open the registry key.

Scope

Specifies the scope of the registry key to open. The scope determines where the information is stored. The key opened can be global or hardware profile-specific. Can be one of the following values:

DICS_FLAG_GLOBAL

Open a key to store global configuration information. This information is not specific to a particular hardware profile. On Windows NT/Windows 2000 this opens a key that is rooted at **HKEY_LOCAL_MACHINE.** The exact key opened depends on the value of the *Key-Type* parameter.

DICS_FLAG_CONFIGSPECIFIC

Open a key to store hardware profile-specific configuration information. This key is rooted at one of the hardware-profile specific branches, instead of **HKEY_LOCAL_MACHINE.** The exact key opened depends on the value of the *KeyType* parameter.

HwProfile

Specifies the hardware profile to open a key for, if *Scope* is set to SPDICS_FLAG_ CONFIGSPECIFIC. If *HwProfile* is 0, the key for the current hardware profile is opened. If *Scope* is SPDICS_FLAG_GLOBAL, this parameter is ignored.

KeyType

Specifies the type of registry storage key to open. Can be one of the following values:

DIREG_DEV

Open a hardware registry key for the device. This is the key for storage of driver-independent configuration information.

This key has the form **HKLM\SYSTEM\CurrentControlSet\Enum\***enumerator*\ *deviceID*\device-instance**\Device Parameters**. Only use this API to open this key. Do not open this registry path directly. This path is only provided here to aid debugging.

DIREG_DRV

Open a software, or driver, registry key for the device.

This key has the form **HKLM\SYSTEM\CurrentControlSet\Control\Class\***ClassGUID*\ *InstanceID* where *classGUID* is the GUID representing the device's class and *InstanceID* is a base-10, four-digit ordinal representing this device instance within the list of device instances for this class. Only use this API to open this key. Do not open this registry path directly. This path is only provided here to aid debugging.

samDesired

Specifies the access you require for this key.

Return Value

If the function is successful, it returns a handle to an opened registry key where private configuration data pertaining to this device instance can be stored/retrieved.

If the function fails, it returns INVALID_HANDLE_VALUE. To get extended error information, call **GetLastError**.

Comments

Close the handle returned from this function by calling **RegCloseKey**.

The specified device instance must be registered before calling this function. After creating the device instance with **SetupDiCreateDeviceInfo**, call **SetupDiRegisterDeviceInfo** to register it.

See Also

SetupDiCreateDeviceInfo, **SetupDiCreateDevRegKey**, **SetupDiGetHwProfileList**, **SetupDiRegisterDeviceInfo**

SetupDiRegisterCoDeviceInstallers

```
BOOLEAN
  SetupDiRegisterCoDeviceInstallers(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData
    );
```

SetupDiRegisterCoDeviceInstallers registers the device-specific coinstallers listed in the INF file for the specified device. This function is the default handler for DIF_REGISTER_COINSTALLERS.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing a device information element for which coinstallers are to be registered. The device information set must not contain any remote elements.

DeviceInfoData

Identifies the device within the device information set. This parameter identifies a device information element in the device information set.

Return Value

SetupDiRegisterCoDeviceInstallers returns TRUE if the function succeeds. If the function returns FALSE, call **GetLastError** for extended error information.

Comments

SetupDiRegisterCoDeviceInstallers reads the INF file for the device specified by *DeviceInfoData* and creates registry entries to register any device-specific coinstallers listed in the INF. Coinstallers are listed in an [*XxxInstallSec*.**CoInstallers**] section, where *XxxInstallSec* is the install section name for the selected driver node, potentially decorated with an OS/architecture-specific extension. This function also copies the files for the coinstallers, unless the DI_NOFILECOPY flag is set.

If there is no driver selected or the device has a legacy INF file, this function does not register any coinstallers.

Registering a new device-specific coinstaller invalidates the Device Installer's current list of coinstallers. After a successful registration, the Device Installer updates its list of co-installers.

This function only registers device-specific coinstallers, not class coinstallers. See the *Plug and Play, Power Management, and Setup Design Guide* for information on registering a class coinstaller.

See the *Plug and Play, Power Management, and Setup Design Guide* for further information on writing device-specific coinstallers.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

DIF_REGISTER_COINSTALLERS, **SetupDiCallClassInstaller**

SetupDiRegisterDeviceInfo

```
BOOLEAN
  SetupDiRegisterDeviceInfo(
    IN HDEVINFO  DeviceInfoSet,
    IN OUT PSP_DEVINFO_DATA  DeviceInfoData,
    IN DWORD  Flags,
    IN PSP_DETSIG_CMPPROC  CompareProc,  OPTIONAL
    IN PVOID  CompareContext,  OPTIONAL
    OUT PSP_DEVINFO_DATA  DupDeviceInfoData  OPTIONAL
    );
```

SetupDiRegisterDeviceInfo registers a newly created device instance with the Plug and Play Manager. This function is the default handler for the DIF_REGISTERDEVICE request and should only be called for nonPnP devices.

Parameters

DeviceInfoSet

Supplies a handle to a device information set. The device information set must not contain any remote elements.

DeviceInfoData

Supplies a pointer to a SP_DEVINFO_DATA structure that identifies the device in the *DeviceInfoSet*. This is an IN OUT parameter because the **DevInst** field of the structure can be updated with a new handle value upon return.

Flags

Controls how the device is registered. Can be the following value:

SPRDI_FIND_DUPS

Search for a previously-existing device instance corresponding to the device information pointed to by *DeviceInfoData*. If this flag is not specified, the device instance is registered regardless of whether or not a device instance already exists for it.

If the caller supplies a *CompareProc* they must also set this flag.

CompareProc

Supplies a comparison callback function to use in duplicate detection. If specified, the function is called for each device instance that is of the same class as the device instance being registered. The prototype of the callback function is as follows:

```
typedef  DWORD (CALLBACK* PSP_DETSIG_CMPPROC) (
    IN HDEVINFO          DeviceInfoSet,
    IN PSP_DEVINFO_DATA  NewDeviceData,
    IN PSP_DEVINFO_DATA  ExistingDeviceData,
    IN PVOID             CompareContext      OPTIONAL
    );
```

The compare function must return ERROR_DUPLICATE_FOUND if it finds that the two devices are duplicates. Otherwise it should return NO_ERROR. If some other error is encountered, the callback function should return the appropriate ERROR_* code to indicate the failure.

If *CompareProc* is not specified and duplication detection is requested, a default comparison behavior is used. The default is to compare the new device's detect signature with the detect signature of all other devices in the class. The detect signature is contained in the class-specific resource descriptor of the device's boot log configuration.

CompareContext

Supplies a pointer to a caller-supplied context buffer that is passed into the callback function. This parameter is ignored if *CompareProc* is not specified.

DupDeviceInfoData

Optionally supplies a pointer to a device information element to receive a duplicate device instance, if any, discovered as a result of attempting to register this device. The caller must set **cbSize** to **sizeof**(SP_DEVINFO_DATA). This will be filled in if the function returns FALSE, and **GetLastError** returns ERROR_DUPLICATE_FOUND. This device information element is added as a member of the specified *DeviceInfoSet*, if not already a member. If *DupDeviceInfoData* is not specified, the duplicate is not added to the device information set.

If you call this function when handling a DIF_REGISTERDEVICE request, the *DupDeviceInfoData* parameter must be NULL.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

Do not call this function for PnP devices. PnP devices are registered by the OS.

After registering a device information element, the caller should refresh any stored copies of the **DevInst** handle associated with this device. This is necessary because the handle value might have changed during registration. The caller need not retrieve the SP_DEVINFO_DATA structure again because the **DevInst** field of the structure is updated to reflect the current value of the handle.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

DIF_REGISTERDEVICE, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

SetupDiRemoveDevice

```
BOOLEAN
  SetupDiRemoveDevice(
    IN HDEVINFO  DeviceInfoSet,
    IN OUT PSP_DEVINFO_DATA  DeviceInfoData
    );
```

SetupDiRemoveDevice is the default handler for the DIF_REMOVE installation request. It removes a device from the system.

Parameters

DeviceInfoSet

Supplies a handle to a device information set for the local machine.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the *DeviceInfoSet*. This is an IN OUT parameter because the **DevInst** field of the structure can be updated with a new handle value upon return. If this is a global removal or the last hardware profile-specific removal, all traces of the device instance are deleted from the registry and the *DeviceInfoSet* handle is NULL.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

Comments

This function removes the device from the system. It deletes the device's hardware and software registry keys and any hardware-profile-specific registry keys (configuration-specific registry keys). This function dynamically stops the device if its **DevInst** is active and this is a global removal or the last configuration-specific removal. If the device cannot be dynamically stopped, flags are set in the Install Parameter block of the device information set that eventually cause the user to be prompted to shut down the system.

Device removal is either global to all hardware profiles or specific to one hardware profile as specified by the **ClassInstallParams** field of the structure. Configuration-specific removal is only appropriate for root-enumerated devices and should only be requested by system code.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SP_DEVINFO_DATA, SP_REMOVEDEVICE_PARAMS

SetupDiRemoveDeviceInterface

```
BOOLEAN
  SetupDiRemoveDeviceInterface(
    IN HDEVINFO  DeviceInfoSet,
    IN OUT PSP_DEVICE_INTERFACE_DATA  DeviceInterfaceData
    );
```

SetupDiRemoveDeviceInterface removes a registered device interface from the system.

Parameters

DeviceInfoSet

Points to the device information set containing the interface and its underlying device. This handle is typically returned by **SetupDiGetClassDevs**. The device information set must not contain remote elements.

DeviceInterfaceData

Points to a structure that identifes the interface being removed. This information is typically returned by **SetupDiEnumDeviceInterfaces**.

After the interface is removed, this function sets the SPINT_REMOVED flag in *Device-InterfaceData*.**Flags**. It also clears the SPINT_ACTIVE flag, but note that this flag should have already been cleared when this function was called.

Return Value

SetupDiRemoveDeviceInterface returns TRUE if the function completed without error. If the function completed with an erorr, it returns FALSE and the error code for the failure can be retrieved by calling **GetLastError**.

Comments

SetupDiRemoveDeviceInterface removes the specified device interface from the system, including deleting the associated registry key.

Call **SetupDiDeleteDeviceInterfaceData** to delete the interface from a device information list.

A device interface must be disabled to be removed. If the interface is enabled, this function fails and **GetLastError** returns ERROR_DEVICE_INTERFACE_ACTIVE. Disable an interface using whatever interface-specific mechanism is provided (for example, an IOCTL). If the caller has no way of disabling an interface and the interface must be removed, the caller must stop the underlying device using **SetupDiChangeState**. Stopping the device disables all the interfaces exposed by the device.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiChangeState, **SetupDiCreateDeviceInterface**, **SetupDiDeleteDeviceInterface-Data**, **SetupDiEnumDeviceInterfaces**, **SetupDiGetClassDevs**

SetupDiSelectBestCompatDrv

```
BOOLEAN
  SetupDiSelectBestCompatDrv(
    IN HDEVINFO DeviceInfoSet,
    IN OUT PSP_DEVINFO_DATA DeviceInfoData
    );
```

SetupDiSelectBestCompatDrv is the default handler for the DIF_ SELECTBESTCOMPATDRV installation request.

Parameters

DeviceInfoSet

Supplies a handle to a device information set. The device information set must not contain any remote elements.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that indicates the member of the device information set for which a driver is to be selected.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

Comments

SetupDiSelectBestCompatDrv selects the best driver for the device from the device information element's compatible driver list. To get the selected driver for a device, call **SetupDiGetSelectedDriver**.

The best driver has the lowest "rank". A "rank zero" match is the best match. To choose between several drivers in the driver list that have the same best-rank match, this function chooses the driver with the most-recent date. A driver's date is set with a **DriverVer** entry in the driver's INF file. This function only considers the **DriverVer** date if the INF is digitally signed.

This function uses information on the local machine when selecting the best driver.

See Also

DIF_SELECTBESTCOMPATDRV, SP_DEVINFO_DATA

SetupDiSelectDevice

```
BOOLEAN
  SetupDiSelectDevice(
    IN HDEVINFO DeviceInfoSet,
    IN OUT PSP_DEVINFO_DATA DeviceInfoData  OPTIONAL
    );
```

SetupDiSelectDevice is the default handler for the DIF_SELECTDEVICE request.

Parameters

DeviceInfoSet

Supplies a handle to a device information set for the local machine.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that indicates the member of the device information set for which a driver is to be selected. If this parameter is not specified, a driver is selected for the global class driver list associated with the device information set.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

Comments

The function handles the user interface that allows the user to select a driver for the device specified. By setting the **Flags** field of the SP_DEVINSTALL_PARAMS structure, the caller can specify special handling of the user interface such as to allow users to select a driver from an OEM installation disk.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiCallClassInstaller, SP_DEVINSTALL_PARAMS

SetupDiSelectOEMDrv

```
BOOLEAN
  SetupDiSelectOEMDrv(
    IN HWND  hwndParent,  OPTIONAL
    IN HDEVINFO  DeviceInfoSet,
    IN OUT PSP_DEVINFO_DATA  DeviceInfoData  OPTIONAL
    );
```

SetupDiSelectOEMDrv selects a driver for a device using an OEM path supplied by the user.

Parameters

hwndParent

Supplies a window handle that will be the parent of any dialogs created during the processing of this function. This parameter can be used to override the **hwndParent** field in the install parameters block of the specified device information set or element. The device information set must not contain remote elements.

DeviceInfoSet

Supplies a handle to the device information set that contains the device being installed.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure for the device being installed. If this parameter is not specified, the device being installed is associated with the global class driver list of the device information set.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

Comments

This function asks the user for the OEM path and then calls the class installer to select a driver from the OEM path.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

SetupDiAskForOEMDisk

SetupDiSetClassInstallParams

```
BOOLEAN
  SetupDiSetClassInstallParams(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN PSP_CLASSINSTALL_HEADER  ClassInstallParams,  OPTIONAL
    IN DWORD  ClassInstallParamsSize
    );
```

SetupDiSetClassInstallParams sets or clears class install parameters for a device information set or a particular device information element.

Parameters

DeviceInfoSet

Supplies a handle to the device information set that contains the class install parameters to set.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that contains the class install parameters to set. If *DeviceInfoData* is not specified, the class install parameters to set are associated with the device information set.

ClassInstallParams

Supplies a pointer to a buffer that contains the new class install parameters to use. The SP_CLASSINSTALL_HEADER structure at the beginning of this buffer must have its **cbSize** field set to **sizeof**(SP_CLASSINSTALL_HEADER) and the **InstallFunction** field must be set to the DI_FUNCTION code that reflects the type of parameters contained in the rest of the buffer.

If *ClassInstallParams* is not specified, the current class install parameters, if any, are cleared for the specified device information set or element.

ClassInstallParamsSize

Supplies the size, in bytes, of the *ClassInstallParams* buffer. If the buffer is not supplied (that is, the class install parameters are being cleared), *ClassInstallParamsSize* must be 0.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

All parameters are validated before any changes are made. Therefore, a return value of FALSE indicates that no parameters were modified.

A consequence of setting class install parameters is that the DI_CLASSINSTALLPARAMS flag is set. If the caller wants to set the parameters, but disable their use, this flag must be cleared by a call to **SetupDiSetDeviceInstallParams**.

If the class install parameters are cleared, the DI_CLASSINSTALLPARAMS flag is reset.

See Also

SetupDiGetClassInstallParams, **SetupDiSetDeviceInstallParams**

SetupDiSetDeviceInstallParams

```
BOOLEAN
  SetupDiSetDeviceInstallParams(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN PSP_DEVINSTALL_PARAMS  DeviceInstallParams
    );
```

SetupDiSetDeviceInstallParams sets device install parameters for a device information set or a particular device information element.

Parameters

DeviceInfoSet

Supplies a handle to the device information set that contains the device install parameters to set.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that contains device install parameters to set. If this parameter is not specified, the install parameters set are associated with the device information set for the global class driver list.

DeviceInstallParams

Supplies a pointer to an SP_DEVINSTALL_PARAMS structure that contains the new values of the parameters. The **cbSize** field of this structure must be set to the size, in bytes, of the structure before calling this function.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

All parameters are validated before any changes are made. Therefore, a return value of FALSE indicates that no parameters were modified.

See Also

SetupDiGetDeviceInstallParams

SetupDiSetDeviceRegistryProperty

```
BOOLEAN
  SetupDiSetDeviceRegistryProperty(
    IN HDEVINFO  DeviceInfoSet,
    IN OUT PSP_DEVINFO_DATA  DeviceInfoData,
    IN DWORD  Property,

IN CONST BYTE  *PropertyBuffer,
    IN DWORD  PropertyBufferSize
    );
```

SetupDiSetDeviceRegistryProperty sets the specified Plug and Play device property.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing information about the device instance whose Plug and Play property is to be modified.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure indicating the device instance whose Plug and Play property is to be modified. If the ClassGUID property is being set, this structure is updated upon return to reflect the device's new class.

Property

Supplies an ordinal specifying the property to be set. Can be one of the following values:

Code	Property
SPDRP_CHARACTERISTICS	Device Characteristics
SPDRP_COMPATIBLEIDS	CompatibleIDs
SPDRP_CONFIGFLAGS	ConfigFlags
SPDRP_DEVTYPE	Device Type
SPDRP_EXCLUSIVE	Exclusive access
SPDRP_FRIENDLYNAME	FriendlyName
SPDRP_HARDWAREID	HardwareID

Code	Property
SPDRP_LOCATION_INFORMATION	LocationInformation
SPDRP_LOWERFILTERS	LowerFilters
SPDRP_SECURITY	Security (binary form)
SPDRP_SECURITY_SDS	Security (SDS form)
SPDRP_SERVICE	Service
SPDRP_UI_NUMBER_DESC_FORMAT	Format-message-style string to format UI number
SPDRP_UPPERFILTERS	UpperFilters

Do not set the following values:

SPDRP_ADDRESS

SPDRP_BUSNUMBER

SPDRP_BUSTYPEGUID

SPDRP_CAPABILITIES

SPDRP_CLASS (established by the class GUID in the INF)

SPDRP_CLASSGUID (established by the class GUID in the INF)

SPDRP_DEVICEDESC (provided by the bus driver or INF)

SPDRP_DRIVER (determined by PnP during device installation)

SPDRP_ENUMERATOR_NAME

SPDRP_LEGACYBUSTYPE

SPDRP_MFG (specified by the INF)

SPDRP_PHYSICAL_DEVICE_OBJECT_NAME

SPDRP_UI_NUMBER

PropertyBuffer

The address of a buffer that contains the new data for the property. If the property is being cleared, then this pointer should be NULL and *PropertyBufferSize* must be zero.

PropertyBufferSize

Supplies the size, in bytes, of *PropertyBuffer*. If *PropertyBuffer* is NULL, then this field must be zero.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

Comments

The class name property cannot be set because it is based on the corresponding class GUID and is automatically updated when that property is changed. When the ClassGUID property changes, **SetupDiSetDeviceRegistryProperty** automatically cleans up any software keys associated with the device.

See Also

SetupDiGetDeviceRegistryProperty

SetupDiSetDriverInstallParams

```
BOOLEAN
  SetupDiSetDriverInstallParams(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN PSP_DRVINFO_DATA  DriverInfoData,
    IN PSP_DRVINSTALL_PARAMS  DriverInstallParams
    );
```

SetupDiSetDriverInstallParams establishes install parameters for the specified driver.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing a driver information structure for which to set installation parameters.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that contains a driver information structure for which to set installation parameters. If this parameter is not specified, the driver referenced is a member of the global class driver list owned by the device information set.

DriverInfoData

Supplies a pointer to an SP_DRVINFO_DATA structure that specifies the driver for which install parameters are to be set.

DriverInstallParams

Supplies a pointer to an SP_DRVINSTALL_PARAMS structure that specifies what the new driver install parameters should be. The **cbSize** field of this structure must be set to the size, in bytes, of the structure before this function is called.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

See Also

SetupDiGetDriverInstallParams

SetupDiSetSelectedDevice

```
BOOLEAN
  SetupDiSetSelectedDevice(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData
    );
```

SetupDiSetSelectedDevice sets the specified device information element to be the currently-selected member of a device information set. This function is typically used by an installation wizard.

Parameters

DeviceInfoSet

Supplies a handle to the device information set for which the selected device should be set.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that specifies the device information element to select.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

See Also

SetupDiGetSelectedDevice

SetupDiSetSelectedDriver

```
BOOLEAN
  SetupDiSetSelectedDriver(
    IN HDEVINFO  DeviceInfoSet,
    IN PSP_DEVINFO_DATA  DeviceInfoData,  OPTIONAL
    IN OUT PSP_DRVINFO_DATA  DriverInfoData  OPTIONAL
    );
```

SetupDiSetSelectedDriver sets the specified member of a driver list as the currently-selected driver. It can also be used to reset the driver list so that there is no currently-selected driver.

Parameters

DeviceInfoSet

Supplies a handle to the device information set containing information about the device instance for which to set a selected driver.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that contains the device information element for which to select a driver. If this parameter is NULL, a class driver for the global class driver list is selected.

DriverInfoData

If this parameter is specified, it supplies a pointer to a driver information structure that indicates the driver to be selected. If this parameter is NULL, the driver list is reset (that is, no driver is selected).

If the **Reserved** field of the SP_DRVINFO_DATA structure is 0, the caller is requesting a search for a driver node with the specified parameters (DriverType, Description, and ProviderName). If a match is found, that driver node is selected. The **Reserved** field is updated on output to reflect the actual driver node where the match was found. If a match is not found, the function fails and a call to **GetLastError** returns ERROR_INVALID_PARAMETER.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved with a call to **GetLastError**.

See Also

SetupDiGetSelectedDriver

SetupDiUnremoveDevice

```
BOOLEAN
  SetupDiUnremoveDevice(
    IN HDEVINFO  DeviceInfoSet,
    IN OUT PSP_DEVINFO_DATA  DeviceInfoData
    );
```

SetupDiUnremoveDevice is the default handler for the DIF_UNREMOVE installation request. This function restores a device to a hardware profile and starts the device, if possible.

Parameters

DeviceInfoSet

Supplies a handle to a device information set for the local machine.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the *DeviceInfoSet*. This is an IN OUT parameter because the **DevInst** field of the structure can be updated with a new handle value upon return.

Return Value

The function returns TRUE if it is successful. Otherwise it returns FALSE and the logged error can be retrieved by a call to **GetLastError**.

Comments

SetupDiUnremoveDevice restores a device to a hardware profile. This function starts the device, if possible, or it sets a flag in the device install parameters that eventually causes the user to be prompted to shut down the system.

The device being restored must have class install parameters for DIF_UNREMOVE or the function fails and **GetLastError** returns ERROR_NO_CLASSINSTALL_PARAMS.

The *DeviceInfoSet* must only contain elements on the local machine.

See Also

DIF_UNREMOVE, *SetupDiRemoveDevice*, SP_DEVINFO_DATA

UpdateDriverForPlugAndPlayDevices

```
BOOLEAN
  UpdateDriverForPlugAndPlayDevices(
    HWND  hwndParent,
    LPCTSTR  HardwareId,
    LPCTSTR  FullInfPath,
    DWORD  InstallFlags,
    PBOOL  bRebootRequired OPTIONAL
    );
```

Given an INF and a hardware ID, **UpdateDriverForPlugAndPlayDevices** installs updated drivers for devices that match the hardware ID.

Parameters

hwndParent

A caller-supplied handle to the top-level window to use for any UI related to installing device(s).

HardwareId

A caller-supplied Hardware ID to match against existing devices on the machine.

FullInfPath

A caller-supplied full path to an INF and any associated driver files. The files should be on the distribution media or in a vendor-created directory, not in a system location such as *%windir%\inf*. The function copies driver files to the appropriate system locations if the installation is successful.

InstallFlags

This parameter is typically zero.

In a special situation, an application might specify the INSTALLFLAG_FORCE flag. If this flag is set and this function finds a device that matches the *HardwareId*, it installs new driver(s) for the device whether or not better drivers already exist on the machine. Vendors must use this flag with extreme caution because it can cause an older driver to be installed over a newer driver.

bRebootRequired

Optional address of a BOOLEAN that indicates whether a reboot is required and who should prompt for it.

If *bRebootRequired* is NULL, this function prompts for a reboot after installing driver(s), if necessary. If *bRebootRequired* is a valid pointer, this function returns its reboot status through this parameter and it is the caller's responsibility to prompt for a reboot if one is needed.

Return Value

The function returns TRUE if a device was upgraded to the specified driver.

Otherwise, it returns FALSE and the logged error can be retrieved with a call to **GetLastError**. Possible error values returned by **GetLastError** include:

Error Value	Description
ERROR_FILE_NOT_FOUND	The *FullInfPath* does not exist.
ERROR_NO_SUCH_DEVINST	The *HardwareId* does not match any device on the machine.
ERROR_INVALID_FLAGS	*InstallFlags* does not contain an accepted value.
NO_ERROR	The routine found a match for the *HardwareId* but the specified driver was not better than the current driver and the caller did not specify the INSTALLFLAG_FORCE flag.

Comments

UpdateDriverForPlugAndPlayDevices scans the devices on the machine and attempts to install the driver(s) specified at *FullInfPath* on any device(s) that match the given *HardwareId*. The default behavior is to only install the specified driver(s) if they are better than the currently installed driver(s) and the specified driver(s) are also better than any driver(s) in *%windir%\inf*. For more information, see *How Does Setup Select a Driver for a Device?* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide*

UpdateDriverForPlugAndPlayDevices attempts to install the specified driver(s) on all hardware that matches the specified *HardwareId*.

This function is defined in *newdev.h*. Applications that call this function must link to *newdev.lib*.

CHAPTER 4

Device Installation Structures

This chapter describes the structures that are parameters to **SetupDi*Xxx*** functions, device installation function requests (DIF requests), and other device installation operations.

SP_ADDPROPERTYPAGE_DATA

```
typedef struct _SP_ADDPROPERTYPAGE_DATA {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    DWORD Flags;
    HPROPSHEETPAGE DynamicPages[MAX_INSTALLWIZARD_DYNAPAGES];
    DWORD NumDynamicPages;
    HWND hwndWizardDlg;
} SP_ADDPROPERTYPAGE_DATA, *PSP_ADDPROPERTYPAGE_DATA;
```

An installer uses an SP_ADDPROPERTYPAGE_DATA structure to supply custom property page(s) for a device when handling a DIF_ADDPROPERTYPAGE_ADVANCED request.

Members

ClassInstallHeader

An install request header that contains the header size and the DIF code for the request.

Flags

Reserved. Must be zero.

DynamicPages

An array of property sheet page handles. An installer can add the handles of custom property pages to this array.

NumDynamicPages

The number of pages added to the **DynamicPages** array.

Because the array index is zero-based, this value is also the index to the next free entry in the array. For example, if there are 3 pages in the array, **DynamicPages**[3] is the next entry for an installer to use.

hwndWizardDlg

The window handle of the Device Manager top-level window.

Comments

See the Platform SDK for documentation on the PROPSHEETPAGE structure and for more information on property pages.

See Also

DIF_ADDPROPERTYPAGE_ADVANCED

SP_CLASSIMAGELIST_DATA

```
typedef struct _SP_CLASSIMAGE_DATA {
    DWORD cbSize;
    HIMAGELIST ImageList;
    DWORD Reserved;
} SP_CLASSIMAGE_DATA, *PSP_CLASSIMAGE_DATA;
```

An SP_CLASSIMAGELIST_DATA structure describes a class image list.

Members

cbSize

The size, in bytes, of the SP_CLASSIMAGE_DATA structure.

ImageList

A handle to the class image list.

See Also

SetupDiDestroyClassImageList, **SetupDiGetClassImageIndex**, **SetupDiGetClass-ImageList**

SP_CLASSINSTALL_HEADER

```
typedef struct _SP_CLASSINSTALL_HEADER {
    DWORD cbSize;
    DI_FUNCTION InstallFunction;
} SP_CLASSINSTALL_HEADER, *PSP_CLASSINSTALL_HEADER;
```

An SP_CLASSINSTALL_HEADER is the first member of any class install parameters structure. It contains the device installation request code that defines the format of the rest of the install parameters structure.

Members

cbSize

The size, in bytes, of the SP_CLASSINSTALL_HEADER structure.

InstallFunction

The device installation request (DIF code) for the class install parameters structure.

DIF codes have the format DIF_*XXX* and are defined in *setupapi.h.* See the chapter on *Device Installation Function Codes* for a complete description of DIF codes.

Comments

When a component allocates a class install parameters structure, it typically initializes the header fields of the structure. Such a component sets the **InstallFunction** member to the DIF code for the installation request and sets **cbSize** to the size of the SP_CLASSINSTALL_HEADER structure. For example:

```
SP_REMOVEDEVICE_PARAMS RemoveDeviceParams;
RemoveDeviceParams.ClassInstallHeader.cbSize = sizeof(SP_CLASSINSTALL_HEADER);
RemoveDeviceParams.ClassInstallHeader.InstallFunction = DIF_REMOVE;
```

A component must set the **InstallFunction** before passing a class install parameters structure to **SetupDiSetClassInstallParams**.

However, a component need not set this field when passing class install parameters to **SetupDiGetClassInstallParams**. This function sets the **InstallFunction** in the structure it passes back to the caller; this function sets it to the DIF_*XXX* code for the currently active device installation request.

See Also

SetupDiCallClassInstaller, **SetupDiGetClassInstallParams**, **SetupDiSetClass-InstallParams**, SP_DETECTDEVICE_PARAMS, SP_MOVEDEV_PARAMS, SP_NEWDEVICEWIZARD_DATA, SP_POWERMESSAGEWAKE_PARAMS, SP_PROPCHANGE_PARAMS, SP_REMOVEDEVICE_PARAMS, SP_SELECTDEVICE_PARAMS, SP_TROUBLESHOOTER_PARAMS, SP_UNREMOVEDEVICE_PARAMS

SP_DETECTDEVICE_PARAMS

```
typedef struct _SP_DETECTDEVICE_PARAMS {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    PDETECT_PROGRESS_NOTIFY DetectProgressNotify;
    PVOID ProgressNotifyParam;
} SP_DETECTDEVICE_PARAMS, *PSP_DETECTDEVICE_PARAMS;
```

An SP_DETECTDEVICE_PARAMS structure corresponds to a DIF_DETECT installation request.

Members

ClassInstallHeader

An install request header that contains the size of the header and the DIF code for the request. See *SP_CLASSINSTALL_HEADER* for more information.

DetectProgressNotify

A callback routine that displays a progress bar for the device detection operation. The callback routine is supplied by the Setup component that sends the DIF_DETECT request. The callback has the following prototype:

```
typedef BOOL (CALLBACK* PDETECT_PROGRESS_NOTIFY)(
    IN PVOID ProgressNotifyParam,
    IN DWORD DetectComplete
    );
```

ProgressNotifyParam is an opaque "handle" that identifies the detection operation. This value is supplied by the Setup component that sent the DIF_DETECT request.

DetectComplete is a value between 0 and 100 that indicates the percent completion. The class installer increments this value at various stages of its detection activities, to notify the user of its progress.

ProgressNotifyParam

The opaque **ProgressNotifyParam** "handle" that the class installer passes to the progress callback routine.

See Also

DIF_DETECT, **SetupDiCallClassInstaller**, SP_CLASSINSTALL_HEADER

SP_DEVICE_INTERFACE_DATA

```
typedef struct _SP_DEVICE_INTERFACE_DATA {
    DWORD cbSize;
    GUID InterfaceClassGuid;
    DWORD Flags;
    ULONG_PTR Reserved;
} SP_DEVICE_INTERFACE_DATA, *PSP_DEVICE_INTERFACE_DATA;
```

An SP_DEVICE_INTERFACE_DATA structure defines a device interface in a device information set.

Members

cbSize

The size, in bytes, of the SP_DEVICE_INTERFACE_DATA structure.

InterfaceClassGuid

The GUID for the class to which the device interface belongs.

Flags

Can be one of the following:

SPINT_ACTIVE

The interface is active (enabled).

SPINT_DEFAULT

Reserved. Not currently used.

SPINT_REMOVED

The interface is removed.

See Also

SetupDiCreateDeviceInterface, **SetupDiEnumDeviceInterfaces**, **SetupDiGet-DeviceInterfaceAlias**, **SetupDiOpenDeviceInterface**, SP_DEVICE_INTERFACE_DETAIL_DATA

SP_DEVICE_INTERFACE_DETAIL_DATA

```
typedef struct _SP_DEVICE_INTERFACE_DETAIL_DATA {
    DWORD cbSize;
    TCHAR DevicePath[ANYSIZE_ARRAY];
} SP_DEVICE_INTERFACE_DETAIL_DATA, *PSP_DEVICE_INTERFACE_DETAIL_DATA;
```

An SP_DEVICE_INTERFACE_DETAIL_DATA structure contains the path for a device interface.

Members

cbSize

The size, in bytes, of the fixed portion of the SP_DEVICE_INTERFACE_DETAIL_DATA structure.

DevicePath

A NULL-terminated string that contains the device interface path. This path can be passed to Win32® functions such as **CreateFile**.

See Also

SetupDiGetDeviceInterfaceDetail

SP_DEVINFO_DATA

```
typedef struct _SP_DEVINFO_DATA {
    DWORD cbSize;
    GUID ClassGuid;
    DWORD DevInst;
    ULONG_PTR Reserved;
} SP_DEVINFO_DATA, *PSP_DEVINFO_DATA;
```

An SP_DEVINFO_DATA structure defines a device instance that is a member of a device information set.

Members

cbSize

The size, in bytes, of the SP_DEVINFO_DATA structure.

ClassGuid

The GUID of the device's setup class.

DevInst

An opaque handle to the device instance (also known as a handle to the devnode).

Some functions, such as **SetupDi***Xxx* functions, take the whole SP_DEVINFO_DATA structure as input to identify a device in a device information set. Other functions, such as CM_*Xxx* functions like *Cm_Get_DevNode_Status*, take this **DevInst** handle as input.

Comments

An SP_DEVINFO_DATA structure identifies a device in a device information set. For example, when Setup sends a DIF_INSTALLDEVICE request to a class installer and coinstallers, it includes a handle to a device information set and a pointer to an SP_DEVINFO_DATA that specifies the particular device. Besides DIF requests, this structure is also used in some **SetupDi***Xxx* functions.

See Also

SP_DEVINFO_LIST_DETAIL_DATA

SP_DEVINFO_LIST_DETAIL_DATA

```
typedef struct _SP_DEVINFO_LIST_DETAIL_DATA {
    DWORD cbSize;
    GUID ClassGuid;
    HANDLE RemoteMachineHandle;
    TCHAR RemoteMachineName[SP_MAX_MACHINENAME_LENGTH];
} SP_DEVINFO_LIST_DETAIL_DATA, *PSP_DEVINFO_LIST_DETAIL_DATA;
```

An SP_DEVINFO_LIST_DETAIL_DATA structure contains information about a device information set, such as its associated setup class GUID (if it has an associated setup class).

Members

cbSize

The size, in bytes, of the SP_DEVINFO_LIST_DETAIL_DATA structure.

ClassGuid

The setup class GUID that is associated with the device information set or GUID_NULL if there is no associated setup class.

RemoteMachineHandle

A Configuration Manager machine handle for the remote machine, if the device information set is for a remote machine. If the device information set is for the local machine, this member is NULL.

This is typically the parameter that components use to access the remote machine. The **RemoteMachineName** contains a string, in case the component requires the name of the remote machine.

RemoteMachineName

A NULL-terminated string that contains the name of the remote machine. If the device information set is for the local machine, this member is an empty string.

See Also

SetupDiGetDeviceInfoListDetail

SP_DEVINSTALL_PARAMS

```
typedef struct _SP_DEVINSTALL_PARAMS {
    DWORD cbSize;
    DWORD Flags;
    DWORD FlagsEx;
    HWND hwndParent;
    PSP_FILE_CALLBACK InstallMsgHandler;
    PVOID InstallMsgHandlerContext;
    HSPFILEQ FileQueue;
    ULONG_PTR ClassInstallReserved;
    DWORD Reserved;
    TCHAR DriverPath[MAX_PATH];
} SP_DEVINSTALL_PARAMS, *PSP_DEVINSTALL_PARAMS;
```

An SP_DEVINSTALL_PARAMS structure contains device installation parameters associated with a particular device information element or associated globally with a device information set.

Members

cbSize

The size, in bytes, of the SP_DEVINSTALL_PARAMS structure.

Flags

Flags that control installation and user interface operations. Some flags can be set prior to sending the device installation request while others are set automatically during the processing of some requests. **Flags** can be a combination of the following values.

The flag values are listed in groups: writeable by setup applications and installers, read only (only set by the OS), reserved, and obsolete. The first group lists flags that are writeable:

DI_CLASSINSTALLPARAMS

Set to use the Class Install parameters. **SetupDiSetClassInstallParams** sets this flag when the caller specifies parameters and clears the flag when the caller specifies a NULL parameters pointer.

DI_COMPAT_FROM_CLASS

Set to force **SetupDiBuildDriverInfoList** to build a device's list of compatible drivers from its class driver list instead of the INF file.

DI_DRIVERPAGE_ADDED

Set by a class installer or coinstaller if the installer supplies a page that replaces the system-supplied driver properties page. If this flag is set, the OS does not display the system-supplied driver page.

DI_DONOTCALLCONFIGMG

Set if the Configuration Manager should not be called to remove or reenumerate devices during the execution of certain device installation functions (for example, **SetupDiInstall-Device**).

Setup applications, including class installers and coinstallers, must obey this flag. If this flag is set, such components must not call the following functions:

```
CM_Reenumerate_DevNode, CM_Reenumerate_DevNode_Ex
CM_Query_And_Remove_SubTree, CM_Query_And_Remove_SubTree_Ex
CM_Setup_DevNode, CM_Setup_DevNode_Ex
CM_Set_HW_Prof_Flags, CM_Set_HW_Prof_Flags_Ex
CM_Enable_DevNode, CM_Enable_DevNode_Ex
CM_Disable_DevNode, CM_Disable_DevNode_Ex
```

DI_ENUMSINGLEINF

Set if installers and other Setup components should only search the INF file specified by SP_DEVINSTALL_PARAMS.**DriverPath**. If this flag is set, **DriverPath** contains the path to a single INF file instead of a path to a directory.

DI_INF_IS_SORTED

Set to indicate that the Select Device page should list drivers in the order they appear in the INF file, instead of sorting them alphabetically.

DI_INSTALLDISABLED

Set if the device should be installed in a disabled state by default. To be recognized, this flag must be set before Setup calls the default handler for the DIF_INSTALLDEVICE request.

DI_NEEDREBOOT

For Windows NT®/Windows® 2000, this flag is set if the device requires that the machine be rebooted after device installation or a device state change. A class installer or coinstaller should set this flag during device installation if the installer determines that a reboot is necessary.

DI_NEEDRESTART

The same as DI_NEEDREBOOT.

DI_NOBROWSE

Set to disable browsing when the user is selecting an OEM disk path. A setup application sets this flag to constrain a user to only installing from the installation media location.

DI_NODI_DEFAULTACTION

Set if **SetupDiCallClassInstaller** should not perform any default action if the class installer returns ERR_DI_DO_DEFAULT or there is not a class installer.

DI_NOFILECOPY

Set if Setup applications and components, such as **SetupDiInstallDevice**, should skip file copying.

DI_NOVCP

Set to disable creation of a new copy queue. Use the caller-supplied copy queue in SP_DEVINSTALL_PARAMS.**FileQueue**.

DI_NOWRITE_IDS

Set to prevent **SetupDiInstallDevice** from writing the INF-specified hardware and compatible IDs to the device properties for the devnode. This flag should only be set for root-enumerated devices.

This flag overrides the DI_FLAGSEX_ALWAYSWRITEIDS flag.

DI_PROPERTIES_CHANGE

Set by the Device Manager if a device's properties have been changed, requiring an update of the installer's user interface.

DI_QUIETINSTALL

Set if the device installer functions must be silent and use default choices wherever possible. Class installers and coinstallers must not display any UI if this flag is set.

DI_RESOURCEPAGE_ADDED

Set by a class installer or coinstaller if the installer supplies a page that replaces the system-supplied resource properties page. If this flag is set, the OS does not display the system-supplied resource page.

DI_SHOWOEM

Set to allow support for OEM disks. If this flag is set, the OS presents a "Have Disk" button on the Select Device page. This flag is set, by default, in system-supplied wizards.

DI_USECI_SELECTSTRINGS

Set if a class installer or coinstaller supplied strings that should be used during **SetupDi-SelectDevice**.

The following flags are read only (only set by the OS):

DI_DIDCLASS

Set if **SetupDiBuildDriverInfoList** has already built a list of the drivers for this class of device. If this list has already been built, it contains all the driver information and this flag is always set. **SetupDiDestroyDriverInfoList** clears this flag when it deletes a list of drivers for a class.

This flag is read only. Only the OS sets this flag.

DI_DIDCOMPAT

Set if **SetupDiBuildDriverInfoList** has already built a list of compatible drivers for this device. If this list has already been built, it contains all the driver information and this flag is always set. **SetupDiDestroyDriverInfoList** clears this flag when it deletes a compatible driver list.

This flag is only set in device installation parameters that are associated with a particular device information element, not in parameters for a device information set as a whole.

This flag is read only. Only the OS sets this flag.

DI_MULTMFGS

Set by **SetupDiBuildDriverInfoList** if a list of drivers for a device setup class contains drivers provided by multiple manufacturers.

This flag is read only. Only the OS sets this flag.

The following flags are reserved:

DI_AUTOASSIGNRES

DI_DISABLED

DI_FORCECOPY

DI_GENERALPAGE_ADDED

DI_OVERRIDE_INFFLAGS

DI_SHOWALL

DI_SHOWCLASS

DI_SHOWCOMPAT

The following flags are obsolete:

DI_NOSELECTICONS

DI_PROPS_NOCHANGEUSAGE

FlagsEx

Additional flags that provide control over installation and user interface operations. Some flags can be set prior to calling the device installer functions while others are set automatically during the processing of some functions. **FlagsEx** can be a combination of the following values.

The flag values are listed in groups: writeable by setup applications and installers, read only (only set by the OS), reserved, and obsolete.

The first group lists flags that are writeable:

DI_FLAGSEX_ALLOWEXCLUDEDDRVS
If set, include drivers that have been marked "Exclude From Select".

For example, if this flag is set, **SetupDiSelectDevice** displays drivers that have the Exclude From Select state and **SetupDiBuildDriverInfoList** includes Exclude From Select drivers in the requested driver list.

A driver is "Exclude From Select" if either it is marked **ExcludeFromSelect** in the INF file or it is a driver for a device whose whole setup class is marked **NoInstallClass** or **No-UseClass** in the class installer INF. Drivers for PnP devices are typically "Exclude From Select"; PnP devices should not be manually installed. To build a list of driver files for a PnP device a caller of **SetupDiBuildDriverInfoList** must set this flag.

DI_FLAGSEX_ALWAYSWRITEIDS
If set and the DI_NOWRITE_IDS flag is clear, always write hardware and compatible IDs to the device properties for the devnode. This flag should only be set for root-enumerated devices.

DI_FLAGSEX_APPENDDRIVERLIST
If set, **SetupDiBuildDriverInfoList** appends a new driver list to an existing list. This flag is relevant when searching multiple locations.

DI_FLAGSEX_DRIVERLIST_FROM_URL
If set, build the driver list from INF(s) retrieved from the URL specified in SP_ DEVINSTALL_PARAMS.**DriverPath**. If the **DriverPath** is an empty string, use the Windows Update web site.

Currently, the OS does not support URLs. Use this flag to direct **SetupDiBuild-DriverInfoLIst** to search the Windows Update web site.

Do not set this flag if DI_QUIETINSTALL is set.

DI_FLAGSEX_EXCLUDE_OLD_INET_DRIVERS
If set, do not include old Internet drivers when building a driver list. This flag should be set any time you are building a list of potential drivers for a device. You can clear this flag if you are just getting a list of drivers currently installed for a device.

DI_FLAGSEX_FILTERCLASSES

If set, **SetupDiBuildClassInfoList** will check for class inclusion filters. This means that a device will not be included in the class list if its class is marked as NoInstallClass.

DI_FLAGSEX_INET_DRIVER

If set, the driver was obtained from the Internet. Setup will not use the device's INF to install future devices because Setup cannot guarantee that it can retrieve the driver files again from the Internet.

DI_FLAGSEX_NO_DRVREG_MODIFY

Do not process the **AddReg** and **DelReg** entries for the device's hardware and software (driver) keys. That is, the **AddReg** and **DelReg** entries in the INF file *DDInstall* and *DDInstall*.**HW** sections.

DI_FLAGSEX_POWERPAGE_ADDED

If set, an installer added their own page for the power properties dialog. The OS will not display the system-supplied power properties page. This flag is only relevant if the device supports power management.

DI_FLAGSEX_PROPCHANGE_PENDING

If set, the user made changes to one or more device property sheets. The property-page provider typically sets this flag.

When the user closes the device property sheet, the Device Manager checks the DI_FLAGSEX_PROPCHANGE_PENDING flag. If it is set, the Device Manager clears this flag, sets the DI_PROPERTIES_CHANGE flag, and sends a DIF_PROPERTYCHANGE request to the installers to notify them that something has changed.

DI_FLAGSEX_SETFAILEDINSTALL

If set, the FAILEDINSTALL flag will be set when **SetupDiInstallDevice** installs a NULL driver.

DI_FLAGSEX_USECLASSFORCOMPAT

Filter INF files on the device's setup class when building a list of compatible drivers. If a device's setup class is known, setting this flag decreases the time required to build a list of compatible drivers when searching INFs that are not precompiled. This flag is ignored if DI_COMPAT_FROM_CLASS is set.

The following flags are read only; only the OS sets these flags:

DI_FLAGSEX_CI_FAILED

Set by the OS if a class installer failed to load or start. This flag is read only.

DI_FLAGSEX_DIDCOMPATINFO

Setup has built a list of driver nodes that are compatible with the device. This flag is read only.

DI_FLAGSEX_DIDINFOLIST

Setup has built a list of driver nodes that includes all the drivers listed in the INFs of the specified setup class. If the specified setup class is NULL because the HDEVINFO set or device has no associated class, the list includes all driver nodes from all available INFs. This flag is read only.

DI_FLAGSEX_IN_SYSTEM_SETUP

If set, installation is occurring during initial system setup. This flag is read only.

The following flags are reserved:

DI_FLAGSEX_BACKUPONREPLACE
DI_FLAGSEX_DEVICECHANGE

DI_FLAGSEX_OLDINF_IN_CLASSLIST
DI_FLAGSEX_PREINSTALLBACKUP
DI_FLAGSEX_USEOLDINFSEARCH

The following flags are obsolete:

DI_FLAGSEX_AUTOSELECTRANK0
DI_FLAGSEX_NOUIONQUERYREMOVE

hwndParent

Window handle that will own the user interface dialogs related to this device.

InstallMsgHandler

Callback used to handle events during file copying. An installer can use a callback, for example, to perform special processing when committing a file queue.

InstallMsgHandlerContext

Private data used by the **InstallMsgHandler** callback.

FileQueue

A handle to a caller-supplied file queue where file operations should be queued but not committed.

If you associate a file queue with a device information set (**SetupDiSetDeviceInstall-Params**), you must disassociate the queue from the device information set before you delete the device information set. If you fail to disassociate the file queue, Setup is not able to decrement its reference count on the device information set and is unable to free the memory.

This queue is only used if the DI_NOVCP flag is set, indicating that file operations should be enqueued but not committed.

ClassInstallReserved

A pointer for class-installer data. Coinstallers must not use this field.

DriverPath

This path is used by the **SetupDiBuildDriverInfoList** function.

See Also

SetupDiBuildClassInfoList, **SetupDiBuildDriverInfoList**, **SetupDiCallClassInstaller**, **SetupDiGetDeviceInstallParams**, **SetupDiInstallDevice**, **SetupDiSelectDevice**, **Setup-DiSetDeviceInstallParams**

SP_DRVINFO_DATA

```
typedef struct _SP_DRVINFO_DATA {
    DWORD cbSize;
    DWORD DriverType;
    ULONG_PTR Reserved;
    TCHAR Description[LINE_LEN];
    TCHAR MfgName[LINE_LEN];
    TCHAR ProviderName[LINE_LEN];
    FILETIME DriverDate;
    DWORDLONG DriverVersion;
} SP_DRVINFOR_DATA, *PSP_DRVINFO_DATA;
```

An SP_DRVINFO_DATA structure contains information about a driver. This structure is a member of a driver information list that can be associated with a particular device instance or globally with a device information set.

Members

cbSize

The size, in bytes, of the SP_DRVINFO_DATA structure.

DriverType

The type of driver represented by this structure. Must be one of the following values:

SPDIT_CLASSDRIVER

This structure represents a class driver.

SPDIT_COMPATDRIVER

This structure represents a compatible driver.

Description

A NULL-terminated string that describes the device supported by this driver.

MfgName

A NULL-terminated string that contains the name of the manufacturer of the device supported by this driver.

ProviderName

A NULL-terminated string giving the provider of this driver. This is typically the name of the organization that creates the driver or INF file. **ProviderName** can be an empty string.

DriverDate

Date of the driver. From the **DriverVer** entry in the INF file. See the *INF DDInstall Section* for more information on the **DriverVer** entry.

DriverVersion

Version of the driver. From the **DriverVer** entry in the INF file.

Comments

This structure equates to SP_DRVINFO_DATA_V2. If you are writing a component that must run on Windows NT and/or Windows 98 as well as Windows 2000, you must use the old version of this structure (V1). To use the old structure, specify the macro USE_SP_ DRVINFO_DATA_V1. Version 1 of the structure contains only the first six members.

See Also

SetupDiEnumDriverInfo, SetupDiGetDriverInstallParams, SetupDiGetSelectedDriver, SetupDiSetDriverInstallParams, SetupDiSetSelectedDriver

SP_DRVINFO_DETAIL_DATA

```
typedef struct _SP_DRVINFO_DETAIL_DATA {
    DWORD cbSize;
    FILETIME InfDate;
    DWORD CompatIDsOffset;
    DWORD CompatIDsLength;
    ULONG_PTR Reserved;
    TCHAR SectionName[LINE_LEN];

  TCHAR InfFileName[MAX_PATH];
    TCHAR DrvDescription[LINE_LEN];
    TCHAR HardwareID[ANYSIZE_ARRAY];
} SP_DRVINFO_DETAIL_DATA, *PSP_DRVINFO_DETAIL_DATA;
```

An SP_DRVINFO_DETAIL_DATA structure contains detailed information about a particular driver information structure.

Members

cbSize

The size, in bytes, of the SP_DRVINFO_DETAIL_DATA structure.

InfDate

Date of the INF file for this driver.

CompatIDsOffset

The offset, in characters, from the beginning of the **HardwareID** buffer where the CompatIDs list begins.

CompatIDsLength

The length, in characters, of the CompatIDs list starting at offset **CompatIDsOffset** from the beginning of the **HardwareID** buffer. The CompatIDs list is a list of NULL-terminated strings with an extra NULL at the end of the list.

SectionName

A NULL-terminated string that contains the name of the INF file *DDINstall* section for this driver. This must be the basic *DDInstall* section name without any OS/architecture-specific extensions; for example, **InstallSec**.

InfFileName

A NULL-terminated string that contains the full-qualified name of the INF file for this driver.

DrvDescription

A NULL-terminated string that describes the driver.

HardwareID

A buffer that contains the HardwareID and Compatible IDs list. This is a list of NULL-terminated strings, with an extra NULL at the end of the list.

See Also

INF DDInstall Section, **SetupDiGetDriverInfoDetail**

SP_DRVINSTALL_PARAMS

```
typedef struct _SP_DRVINSTALL_PARAMS {
    DWORD cbSize;
    DWORD Rank;
    DWORD Flags;
    DWORD_PTR PrivateData;
    DWORD Reserved;
} SP_DRVINSTALL_PARAMS, *PSP_DRVINSTALL_PARAMS;
```

An SP_DRVINSTALL_PARAMS structure contains driver installation parameters associated with a particular driver information element.

Members

cbSize

The size, in bytes, of the SP_DRVINSTALL_PARAMS structure.

Rank

The rank match of this driver. Ranges from 0 to *n*, where 0 is the most compatible.

Flags

Flags that control functions operating on this driver. Can be a combination of the following:

DNF_BAD_DRIVER

Do not use this driver. Installers can read and write this flag.

If this flag is set, **SetupDiSelectBestCompatDrv** and **SetupDiSelectDevice** ignore this driver.

A class installer or coinstaller can set this flag to prevent Setup from listing the driver in the Select Driver dialog box. An installer might set this flag when handling a DIF_SELECT-DEVICE or DIF_SELECTBESTCOMPATDRV request, for example.

DNF_CLASS_DRIVER

This driver is a class driver. This flag is READONLY to installers.

DNF_COMPATIBLE_DRIVER

This driver is a compatible driver. This flag is READONLY to installers.

DNF_DUPDESC

There are other providers supplying drivers that have the same description as this driver. This flag is READONLY to installers.

DNF_DUPPROVIDER

There are other providers supplying drivers that have the same description as this driver. The only difference between this driver and its match is the driver date. This flag is READONLY to installers.

If this flag is set, Setup displays the driver date and driver version next to the driver so the user can distinguish it from its match.

DNF_EXCLUDEFROMLIST

Do not display this driver in any driver-select dialogs.

DNF_INDEXED_DRIVER

Reserved.

DNF_INET_DRIVER

This driver came from the Internet or from Windows Update. This flag is READONLY to installers.

If you call SetupCopyOEMInf you must specify the SPOST_URL flag so that when Setup copies this INF into the *%windir%\inf* directory Setup will mark it as an Internet INF. If you omit this step then Setup will attempt to use this device to install other devices. The resulting problem is that Setup does not have the source files any more and will end up prompting the user with an invalid path.

DNF_LEGACYINF

This driver comes from a legacy INF file. This flag is valid for Windows NT/Windows 2000 only. This flag is READONLY to installers.

DNF_NODRIVER

Set if no physical driver is to be installed for this logical driver.

DNF_OLD_INET_DRIVER

This driver came from the Internet, but Setup does not currently have access to its source files. This flag is READONLY to installers.

The system will not install a driver marked with this flag because Setup does not have the source files and would end up prompting the user with an invalid path. The INF for such a driver can be used for everything except for installing devices.

DNF_OLDDRIVER

This driver presently/previously controlled the associated device. This flag is READONLY to installers.

PrivateData

A field a class installer can use to store private data. Coinstallers should not use this field.

See Also

SetupDiGetDriverInstallParams, **SetupDiSetDriverInstallParams**

SP_ENABLECLASS_PARAMS

```
typedef struct _SP_ENABLECLASS_PARAMS {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    GUID ClassGuid;
    DWORD EnableMessage;
} SP_ENABLECLASS_PARAMS, *PSP_ENABLECLASS_PARAMS;
```

This structure is obsolete.

SP_INSTALLWIZARD_DATA

This structure is obsolete.

Instead of DIF_INSTALLWIZARD, Setup uses the DIF_NEWDEVICEWIZARD_*XXX* requests such as DIF_NEWDEVICEWIZARD_FINISHINSTALL.

SP_MOVEDEV_PARAMS

```
typedef struct _SP_MOVEDEV_PARAMS {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    SP_DEVINFO_DATA SourceDeviceInfoData;
} SP_MOVEDEV_PARAMS, *PSP_MOVEDEV_PARAMS;
```

This structure and its associated DIF_MOVEDEVICE installation request are reserved for system use.

SP_NEWDEVICEWIZARD_DATA

```
typedef struct _SP_NEWDEVICEWIZARD_DATA {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    DWORD Flags;

HPROPSHEETPAGE DynamicPages[MAX_INSTALLWIZARD_DYNAPAGES];
    DWORD NumDynamicPages;
    HWND hwndWizardDlg;
} SP_NEWDEVICEWIZARD_DATA, *PSP_NEWDEVICEWIZARD_DATA;
```

An SP_NEWDEVICEWIZARD_DATA structure is used by installers to extend the operation of the hardware installation wizard by adding custom pages. It is used with DIF_NEWDEVICEWIZARD_*XXX* installation requests.

Members

ClassInstallHeader

An install request header that contains the header size and the DIF code for the request. See *SP_CLASSINSTALL_HEADER* for more information.

Flags

Reserved. Must be zero.

DynamicPages

An array of property sheet page handles. An installer can add the handles of custom wizard pages to this array.

NumDynamicPages

The number of pages added to the **DynamicPages** array.

Because the array index is zero-based, this value is also the index to the next free entry in the array. For example, if there are 3 pages in the array, **DynamicPages**[3] is the next entry for an installer to use.

hwndWizardDlg

The window handle of the hardware installation wizard top-level window.

See Also

DIF_NEWDEVICEWIZARD_FINISHINSTALL, DIF_NEWDEVICEWIZARD_POSTANALYZE, DIF_NEWDEVICEWIZARD_PREANALYZE, DIF_NEWDEVICEWIZARD_PRESELECT, DIF_NEWDEVICEWIZARD_SELECT, SP_CLASSINSTALL_HEADER

SP_POWERMESSAGEWAKE_PARAMS

```
typedef struct _SP_POWERMESSAGEWAKE_PARAMS {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    TCHAR PowerMessageWake[LINE_LEN];
} SP_POWERMESSAGEWAKE_PARAMS, *PSP_POWERMESSAGEWAKE_PARAMS;
```

An SP_POWERMESSAGEWAKE_PARAMS structure corresponds to a DIF_POWERMESSAGEWAKE installation request.

Members

ClassInstallHeader

An install request header that contains the header size and the DIF code for the request. See *SP_CLASSINSTALL_HEADER*.

PowerMessageWake

Buffer that contains a string of custom text. Setup displays this text on the power management page of the device properties display in Device Manager.

Comments

Setup only sends the DIF_POWERMESSAGEWAKE request if the drivers for the device support power management.

See Also

DIF_POWERMESSAGEWAKE, SP_CLASSINSTALL_HEADER

SP_PROPCHANGE_PARAMS

```
typedef struct _SP_PROPCHANGE_PARAMS {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    DWORD StateChange;
    DWORD Scope;
    DWORD HwProfile;
} SP_PROPCHANGE_PARAMS, *PSP_PROPCHANGE_PARAMS;
```

An SP_PROPCHANGE_PARAMS structure corresponds to a DIF_PROPERTYCHANGE installation request.

Members

ClassInstallHeader

An install request header that contains the header size and the DIF code for the request. See *SP_CLASSINSTALL_HEADER*.

StateChange

State change action. Can be one of the following values:

DICS_ENABLE

The device is being enabled.

For this state change, Setup enables the device if the **DICS_FLAG_GLOBAL** flag is specified.

If the **DICS_FLAG_CONFIGSPECIFIC** flag is specified and the current hardware profile is specified then Setup enables the device. If the **DICS_FLAG_CONFIGSPECIFIC** is specified and not the current hardware profile then Setup sets some flags in the registry and does not change the device's state. Setup will change the device state when the specified profile becomes the current profile.

DICS_DISABLE

The device is being disabled.

For this state change, Setup disables the device if the **DICS_FLAG_GLOBAL** flag is specified.

If the **DICS_FLAG_CONFIGSPECIFIC** flag is specified and the current hardware profile is specified then Setup disables the device. If the **DICS_FLAG_CONFIGSPECIFIC** is specified and not the current hardware profile then Setup sets some flags in the registry and does not change the device's state.

DICS_PROPCHANGE

The properties of the device have changed.

For this state change, Setup ignores the **Scope** information and stops and restarts the device.

DICS_START

The device is being started (if the request is for the currently active hardware profile).

DICS_START must be **DICS_FLAG_CONFIGSPECIFIC**; you cannot perform that change globally.

Setup only starts the device if the current hardware profile is specified, otherwise Setup sets a registry flag and does not change the state of the device.

DICS_STOP

The device is being stopped. The driver stack will be unloaded and the CSCONFIGFLAG_DO_NOT_START flag will be set for the device.

DICS_STOP must be **DICS_FLAG_CONFIGSPECIFIC**; you cannot perform that change globally.

Setup only stops the device if the current hardware profile is specified, otherwise Setup sets a registry flag and does not change the state of the device.

Components should not specify DICS_STOP or DICS_START. Instead, they should use DICS_PROPCHANGE to stop and restart a device to cause changes in the device's configuration to take effect.

Scope

Flags that specify the scope of a device property change. Can be one of the following:

DICS_FLAG_GLOBAL

Make the change in all hardware profiles.

DICS_FLAG_CONFIGSPECIFIC

Make the change in the specified profile only.

The following flag is obsolete:

DICS_FLAG_CONFIGGENERAL

HwProfile

Supplies the hardware profile ID for profile-specific changes. Zero specifies the current hardware profile.

See Also

DIF_PROPERTYCHANGE, **SetupDiCallClassInstaller**, **SetupDiChangeState**, SP_CLASSINSTALL_HEADER

SP_PROPSHEETPAGE_REQUEST

```
typedef struct _SP_PROPSHEETPAGE_REQUEST {
    DWORD CbSize;
    DWORD PageRequested;
    HDEVINFO DeviceInfoSet;
    PSP_DEVINFO_DATA DeviceInfoData;
} SP_PROPSHEETPAGE_REQUEST, *PSP_PROPSHEETPAGE_REQUEST;
```

An SP_PROPSHEETPAGE_REQUEST structure can be pa414sed as the first parameter (*lpv*) to the **ExtensionPropSheetPageProc** entry point in the Setupapi DLL. **ExtensionPropSheetPageProc** is used to retrieve a handle to a specified property sheet page. For information on **ExtensionPropSheetPageProc** and related functions, see the Platform SDK documentation.

Members

CbSize

The size, in bytes, of the SP_PROPSHEETPAGE_REQUEST structure.

PageRequested

The property sheet page to add to the to property sheet. Can be one of the following values:

SPPSR_SELECT_DEVICE_RESOURCES

Specifies the Resource Selection page supplied by the Setupapi DLL.

SPPSR_ENUM_BASIC_DEVICE_PROPERTIES

Specifies a page that is supplied by the device's BasicProperties32 provider. That is, an installer or other component that supplied page(s) in response to a DIF_ADDPROPERTYPAGE_BASIC installation request.

SPPSR_ENUM_ADV_DEVICE_PROPERTIES

Specifies a page that is supplied by the class and/or the device's EnumPropPages32 provider. That is, an installer or other component that supplied page(s) in response to a DIF_ADDPROPERTYPAGE_ADVANCED installation request.

DeviceInfoSet

The handle for the device information set that contains the device being installed.

DeviceInfoData

A pointer to an SP_DEVINFO_DATA structure for the device being installed.

Comments

The component that is retrieving the property pages calls Setupapi's **ExtensionPropSheetPageProc** function and passes in a pointer to a SP_PROPSHEETPAGE_REQUEST structure, the address of their **AddPropSheetPageProc** function, and some private data. The property sheet provider calls the **AddPropSheetPageProc** routine for each property sheet it provides.

The following code excerpt illustrates how to retrieve one page, the Setupapi's Resource Selection page:

```
{
    DWORD Err;
    HINSTANCE hLib;
    FARPROC PropSheetExtProc;
    HPROPSHEETPAGE hPages[2];
    .
    .
    .
       if(!(hLib = GetModuleHandle(TEXT("setupapi.dll")))) {
            return GetLastError();
       }

       if(!(PropSheetExtProc = GetProcAddress(hLib,
                 "ExtensionPropSheetPageProc"))) {
            Err = GetLastError();
            FreeLibrary(hLib);
            return Err;
        }

        PropPageRequest.cbSize = sizeof(SP_PROPSHEETPAGE_REQUEST);
        PropPageRequest.PageRequested  =
            SPPSR_SELECT_DEVICE_RESOURCES;
        PropPageRequest.DeviceInfoSet  = DeviceInfoSet;
        PropPageRequest.DeviceInfoData = DeviceInfoData;
        if(!PropSheetExtProc(&PropPageRequest,
                AddPropSheetPageProc, &hPages[1])) {
            Err = ERROR_INVALID_PARAMETER;
            FreeLibrary(hLib);
            return Err;
        }
        .
        .
        .
}
```

The **AddPropSheetPageProc** for the above excerpt would be something like the following:

```
BOOL
CALLBACK
AddPropSheetPageProc(
    IN HPROPSHEETPAGE hpage,
    IN LPARAM lParam
   )
{
    *((HPROPSHEETPAGE *)lParam) = hpage;
    return TRUE;
}
```

See Also

DIF_ADDPROPERTYPAGE_ADVANCED, DIF_ADDPROPERTYPAGE_BASIC

SP_REMOVEDEVICE_PARAMS

```
typedef struct _SP_REMOVEDEVICE_PARAMS {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    DWORD Scope;
    DWORD HwProfile;
} SP_REMOVEDEVICE_PARAMS, *PSP_REMOVEDEVICE_PARAMS;
```

An SP_REMOVEDEVICE_PARAMS structure corresponds to the DIF_REMOVE installation request.

Members

ClassInstallHeader

An install request header that contains the header size and the DIF code for the request. See *SP_CLASSINSTALL_HEADER*.

Scope

Flags that indicate the scope of the device removal. Can be one of the following values:

DI_REMOVEDEVICE_GLOBAL

Make this change in all hardware profiles. Remove information about the device from the registry.

DI_REMOVEDEVICE_CONFIGSPECIFIC

Make this change to only the hardware profile specified by **HwProfile**. this flag only applies to root-enumerated devices. When Setup removes the device from the last hardware profile in which it was configured, Setup performs a global removal.

HwProfile

The hardware profile ID for profile-specific changes. Zero specifies the current hardware profile.

See Also

DIF_REMOVE, **SetupDiCallClassInstaller**, **SetupDiRemoveDevice**, SP_ CLASSINSTALL_HEADER

SP_SELECTDEVICE_PARAMS

```
typedef struct _SP_SELECTDEVICE_PARAMS {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    TCHAR Title[MAX_TITLE_LEN];
    TCHAR Instructions[MAX_INSTRUCTION_LEN];
    TCHAR ListLabel[MAX_LABEL_LEN];
    TCHAR SubTitle[MAX_SUBTITLE_LEN];
} SP_SELECTDEVICE_PARAMS, *PSP_SELECTDEVICE_PARAMS;
```

An SP_SELECTDEVICE_PARAMS structure corresponds to a DIF_SELECTDEVICE installation request.

Members

ClassInstallHeader

An install request header that contains the header size and the DIF code for the request. See *SP_CLASSINSTALL_HEADER*.

Title

Buffer that contains an installer-provided window title for driver-selection windows. Setup uses this title for the select-driver window header title in the Add/Remove Hardware wizard or the window title for the Select Device dialogs.

Instructions

Buffer that contains an installer-provided select-device instructions.

ListLabel

Buffer that contains an installer-provided label for the list of drivers from which the user can select.

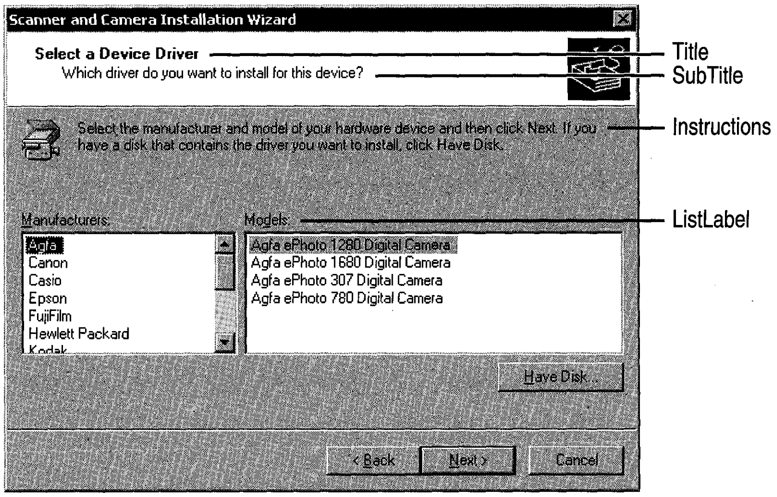
SubTitle

Buffer that contains an installer-provided subtitle used in select-device wizards. This string is not used in select dialogs.

Comments

If an installer sets fields in this structure to be used during driver selection, the installer must also set the DI_USECI_SELECTSTRINGS flag in the SP_DEVINSTALL_PARAMS.

Figure 4.1 shows a sample Select Device dialog box and identifies the strings an installer can supply.



Scanner and Camera Installation Wizard

Select a Device Driver

Which driver do you want to install for this device?

Select the manufacturer and model of your hardware device and then click Next. If you have a disk that contains the driver you want to install, click Have Disk.

Manufacturers:

Agfa
Canon
Casio
Epson
FujiFilm
Hewlett Packard
Kodak

Models:

Agfa ePhoto 1280 Digital Camera
Agfa ePhoto 1680 Digital Camera
Agfa ePhoto 307 Digital Camera
Agfa ePhoto 780 Digital Camera

Figure 4.1 Sample Select Device Dialog

See Also

DIF_SELECTDEVICE, **SetupDiCallClassInstaller**, **SetupDiSelectDevice**, SP_CLASS-INSTALL_HEADER

SP_TROUBLESHOOTER_PARAMS

```
typedef struct _SP_TROUBLESHOOTER_PARAMS {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    TCHAR ChmFile[MAX_PATH];
    TCHAR HtmlTroubleShooter[MAX_PATH];
} SP_TROUBLESHOOTER_PARAMS, *PSP_TROUBLESHOOTER_PARAMS;
```

An SP_TROUBLESHOOTER_PARAMS structure corresponds to a DIF_ TROUBLESHOOTER installation request.

Members

ClassInstallHeader

An install request header that contains the header size and the DIF code for the request. See *SP_CLASSINSTALL_HEADER*.

ChmFile

Optionally specifies a string buffer that contains the path to a CHM file. The CHM file contains HTML help topics with troubleshooting information. The path must be fully qualified if the file is not in default system help directory (%windir%\help).

HtmlTroubleShooter

Optionally specifies a string buffer that contains the path to a topic in the **ChmFile**. This parameter identifies the page of the **ChmFile** that Setup should display first.

Comments

An installer fills in this structure in response to a DIF_TROUBLESHOOTER request.

See Also

DIF_TROUBLESHOOTER, **SetupDiCallClassInstaller**, SP_CLASSINSTALL_HEADER

SP_UNREMOVEDEVICE_PARAMS

```
typedef struct _SP_UNREMOVEDEVICE_PARAMS {
    SP_CLASSINSTALL_HEADER ClassInstallHeader;
    DWORD Scope;
    DWORD HwProfile;
} SP_UNREMOVEDEVICE_PARAMS, *PSP_UNREMOVEDEVICE_PARAMS;
```

An SP_UNREMOVEDEVICE_PARAMS structure corresponds to a DIF_UNREMOVE installation request.

Members

ClassInstallHeader

An install request header that contains the header size and the DIF code for the request. See *SP_CLASSINSTALL_HEADER*.

Scope

A flag that indicates the scope of the unremove operation. This flag must always be set to DI_UNREMOVEDEVICE_CONFIGSPECIFIC

HwProfile

The hardware profile ID for profile-specific changes. Zero specifies the current hardware profile.

See Also

DIF_UNREMOVE **SetupDiCallClassInstaller**, **SetupDiUnremoveDevice** SP_ CLASSINSTALL_HEADER

C H A P T E R 5

Device Installation Function Codes

This chapter describes the Microsoft® Windows® 2000 device installation requests that Setup applications send to class installers and coinstallers. Each request is represented by a Device Installation Function code (a DIF code). The DIF code constants are defined in the *setupapi.h* header file.

Installers that handle these requests include class installers, class coinstallers, and device coinstallers. Some installers are provided by Microsoft and some are provided by OEMs and third-party vendors. Setup sends a DIF request to an installer by calling the installer at its entry point. The DIF code is one of the parameters to the installer routine; other parameters provide additional input.

This chapter describes the DIF codes in alphabetical order.

See *Writing a Coinstaller* and *Writing a Class Installer* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide* for information on writing installers.

DIF_ADDPROPERTYPAGE_ADVANCED

A DIF_ADDPROPERTYPAGE_ADVANCED request allows an installer to supply custom property page(s) for a device.

When Sent

When a user clicks on the properties for a device in the Device Manager or in a Control Panel applet.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Can handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Optionally supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set. If *DeviceInfoSet* is NULL, Setup is requesting property pages for the device setup class.

Device Installation Parameters

Device installation parameters (SP_DEVINSTALL_PARAMS) are associated with the *DeviceInfoData*, if specified, or with the *DeviceInfoSet*.

Class Installation Parameters

An SP_ADDPROPERTYPAGE_DATA structure is associated with the *DeviceInfoData*, if specified, or with the *DeviceInfoSet*.

Output

Device Installation Parameters

An installer can modify the device installation parameters.

Class Installation Parameters

An installer can modify the SP_ADDPROPERTYPAGE_DATA to supply custom page(s).

Return Value

A coinstaller can return NO_ERROR or a Win32® error. A coinstaller should not return ERROR_DI_POSTPROCESSING_REQUIRED for this DIF request.

A class installer returns NO_ERROR if it successfully supplies page(s). Otherwise, a class installer returns ERROR_DI_DO_DEFAULT or a Win32 error.

Default Handler

None.

Operation

In response to this DIF request an installer can supply custom property pages. Installers should handle this DIF request instead of supplying an **EnumPropPages32** registry entry. The **EnumPropPages32** method is supported, but handling this DIF request allows you to supply property pages from a class installer or coinstaller and removes the need for a separate property-page provider.

An installer typically handles this DIF request to add a new device-specific or setup class–specific property page. An installer can also replace the system-supplied driver property page, resource property page, or power property page for a device. If an installer replaces a system-supplied page, the installer must set the appropriate flag in the device installation parameters for the device:

DI_DRIVERPAGE_ADDED

The installer supplied a driver property page.

DI_RESOURCEPAGE_ADDED

The installer supplied a resource property page.

DI_FLAGSEX_POWERPAGE_ADDED

The installer supplied a power property page.

An installer cannot replace the system-supplied general properties page.

Setup only displays one driver page, one resource page, and one power page for a device. An installer should not supply a replacement system page if a previous installer already supplied a page of that type. This constraint does not apply to non-system-supplied property pages.

The following pseudo-code shows how an installer adds a custom page to the array of pages in the class install parameters:

```
:
// get the class install parameters by calling
// SetupDiGetClassInstallParams

// check whether NumDynamicPages has reached the max

// for a system page, check whether a previous installer
// supplied it

// fill in the PROPSHEETPAGE structure

// add the page and increment the NumDynamicPages counter
PropPageData.DynamicPages[PropPageData.NumDynamicPages++] =
    CreatePropertySheetPage(&Page);

// apply the modified parameters by calling
// SetupDiSetClassInstallParams

// for a system page, set the appropriate flag in the devinstall
// parameters
:
```

If an installer adds custom property page(s), the installer should first check whether **NumDynamicPages** in the class install parameters has reached MAX_INSTALLWIZARD_DYNAPAGES.

A coinstaller should add custom pages in its preprocessing pass.

If an installer allows a user to set a property that requires Setup to remove and restart the device, the installer must set the DI_FLAGSEX_PROPCHANGE_PENDING flag in the device installation parameters from its DialogProc routine.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion of handling DIF codes in an installer.

See Also

SP_ADDPROPERTYPAGE_DATA, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

DIF_ALLOW_INSTALL

A DIF_ALLOWINSTALL request asks the installers for a device whether Setup can proceed to install the device.

When Sent

After selecting a driver for the device but before installing the device.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Should not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData*.

Class Installation Parameters

None.

Output

None.

Return Value

A coinstaller can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error.

Typical Win32 error codes for this DIF request include ERROR_DI_DONT_INSTALL, ERROR_NON_WINDOWS_NT_DRIVER, and ERROR_REQUIRES_INTERACTIVE_WINDOWSTATION.

Default Handler

None.

Operation

In response to a DIF_ALLOW_INSTALL request an installer confirms whether Setup can install the device.

An installer can fail this request if it determines that the selected driver is incorrect (for example, if the driver is a Windows 9x-only driver that will not work correctly on Windows 2000) or if it determines that a selected driver is known to have bugs.

An installer might fail this request if the DI_QUIETINSTALL flag is set in the device installation parameters and the installer needs to display UI during device installation. This failure is rare, however, because an installer can typically supply any UI pages in response to the DIF_NEWDEVICEWIZARD_FINISHINSTALL request. In that case, UI does not prevent the installer from succeeding a DIF_ALLOW_INSTALL request for which the quiet flag is set. If, however, an installer cannot limit its UI to the finish-install case, the installer must fail this DIF request if the DI_QUIETINSTALL flag is set. An installer might have this restriction, for example, if it calls third-party code that displays UI.

If an installer fails this DIF request, Setup aborts the installation.

If an installer fails this DIF request and DI_QUIETINSTALL is not set in the device installation parameters, the installer should display a dialog box with a message that explains why the device is not being installed.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

DIF_DESTROYPRIVATEDATA

A DIF_DESTROYPRIVATEDATA request directs a class installer to free any memory or resources it allocated and stored in the **ClassInstallReserved** field of the SP_ DEVINSTALL_PARAMS structure.

When Sent

When Setup destroys a device information set or an SP_DEVINFO_DATA element, or when Setup discards its list of coinstallers and class installer for a device.

Who Handles

Class Coinstaller	Does not handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to a device information set.

DeviceInfoData

Optionally supplies a pointer to an SP_DEVINFO_DATA structure that identifies a device in the device information set.

Device Installation Parameters

Device installation parameters (SP_DEVINSTALL_PARAMS) are associated with the *DeviceInfoData*, if specified, or with the *DeviceInfoSet*.

Class Installation Parameters

None.

Output

Device Installation Parameters

An installer can clear the **ClassInstallReserved** field in the device installation parameters (SP_DEVINSTALL_PARAMS).

Return Value

A coinstaller does not handle this DIF request; it simply returns NO_ERROR in its pre-processing pass.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error.

Default Handler

None.

Operation

In response to a DIF_DESTROYPRIVATEDATA request a class installer frees any memory or resources it allocated and stored in the **ClassInstallReserved** field of the SP_DEVINSTALL_PARAMS structure.

Coinstallers should not use the **ClassInstallReserved** field.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

DIF_DETECT

A DIF_DETECT request directs an installer to detect nonPnP devices of a particular class and add the devices to the device information set. This request is used for nonPnP devices.

When Sent

When the Add/Remove Hardware wizard is detecting nonPnP devices.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set.

Associated Class?

There is a device setup class associated with the *DeviceInfoSet.*

DeviceInfoData

None.

Device Installation Parameters

There are device installation parameters associated with the *DeviceInfoSet.*

Class Installation Parameters

An SP_DETECTDEVICE_PARAMS structure is associated with the *DeviceInfoSet.* The parameters contain a callback routine that the class installer calls to indicate the progress of the detection operation.

Output

DeviceInfoSet

An installer adds a device information element to the *DeviceInfoSet* for each device it detects, regardless of whether a device has been previously detected and installed.

Device Installation Parameters

An installer can modify the device installation parameters for the *DeviceInfoSet* or for new device information elements it creates.

Return Value

If a coinstaller does not detect devices, it returns NO_ERROR from its preprocessing pass. If a coinstaller detects devices, it can do so during pre- or postprocessing and return NO_ERROR or a Win32 error.

If a class installer detects devices, it returns NO_ERROR or an appropriate Win32 error. If a class installer does not handle this DIF request, it returns ERROR_DI_DO_DEFAULT.

Default Handler

None.

Operation

In response to a DIF_DETECT request an installer can detect devices of its setup class.

If an installer detects devices, it should do at least the following:

- Call the **DetectProgressNotify** callback routine in the SP_DETECTDEVICE_PARAMS class installation parameters, if detection will potentially take a noticeable amount of time.

- For each device the installer detects, it should:
  - Create a device information element (**SetupDiCreateDeviceInfo**).
  - Provide information for driver selection.

    The installer can manually select the driver for the device or the installer can set the device's hardware ID that Setup will use to find an INF for the device. An installer sets the hardware ID by calling **SetupDiSetDeviceRegistryProperty** with a *Property* of **SPDRP_HARDWAREID**.
  - Possibly set some device installation parameters.
- Return NO_ERROR for successful detection or return a Win32 error.

If one or more installers detects device(s) in response to this DIF code, Setup compares the list of detected devices to its current list of devices. If the installers detected a new device, Setup attempts to install the device. If the installers omitted a device that appears in Setup's list, Setup typically removes the device.

To detect nonPnP devices during GUI-mode setup, an installer must handle the DIF_FIRSTTIMESETUP request. GUI-mode setup does not send a DIF_DETECT request.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

DIF_FIRSTTIMESETUP, **SetupDiCreateDeviceInfo**, SP_DETECTDEVICE_PARAMS, SP_DEVINSTALL_PARAMS

DIF_INSTALLDEVICE

A DIF_INSTALLDEVICE request allows an installer to perform any final tasks before and/or after the device is installed.

When Sent

After selecting the driver, registering any device coinstallers, and registering any device interfaces.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Can handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device to be installed.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure for the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData*.

Class Installation Parameters

None.

Output

Device Installation Parameters

An installer can modify the device installation parameters for the *DeviceInfoData*. For example, an installer might set the DI_NEEDREBOOT flag or it might set the DI_DONOTCALLCONFIGMG flag to prevent Setup from bringing the device online dynamically with its newly installed driver and settings.

Return Value

A coinstaller typically returns NO_ERROR or ERROR_DI_POSTPROCESSING_REQUIRED. A coinstaller might return a Win32 error.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error. If a class installer successfully installs the device, including superceding all the operations of the default handler, the class installer returns NO_ERROR.

Default Handler

SetupDiInstallDevice

Operation

In response to a DIF_INSTALLDEVICE request an installer typically performs any final installation operations before the default handler installs the device. For example, an installer can check, and possibly modify, the upper-filter drivers and lower-filter drivers for the device listed in the registry.

Unless the DI_NOFILECOPY flag is set in the device installation parameters, an installer that handles this DIF request should copy files required for the device, such as driver files and control panel files.

If the DI_NOFILECOPY flag is clear but the DI_NOVCP flag is set, the installer must enqueue any file operations to the supplied file queue but must not commit the queue.

A coinstaller can handle this DIF request in its preprocessing pass and/or in its post-processing pass. In its preprocessing pass, a coinstaller performs any operations that must occur before Setup loads the drivers and starts the device. In its postprocessing pass, the device is up and running (unless the DI_NEEDREBOOT flag was set and thus Setup could not bring the device online dynamically).

If a class installer performs a few operations but does not replace the default handler, the class installer returns ERROR_DI_DO_DEFAULT.

If this DIF code completes with a Win32 error, Setup aborts the installation.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

DIF_INSTALLDEVICEFILES, **SetupDiInstallDevice**, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

DIF_INSTALLDEVICEFILES

A DIF_INSTALLDEVICEFILES request allows an installer to participate in copying the files to support a device or to make a list of the files for a device. The device files include files for the selected driver, any device interfaces, and any coinstallers.

When Sent

Setup components send this DIF request for a variety of reasons. Some Setup components send this DIF request before DIF_REGISTER_COINSTALLERS, DIF_INSTALLINTERFACES, and DIF_INSTALL_DEVICE to ensure that all the relevant files can be copied before proceeding with the installation. Some Setup components omit this DIF request and expect the files to be copied during the handling of those three DIF requests. In addition, some Setup components send this DIF request to retrieve the list of the files associated with a device.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device whose supporting files are to be copied.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData.*

If the DI_NOVCP flag is set, the device installation parameters contain a valid **FileQueue** handle and installers that handle this DIF request add their file operations to this queue and do not commit the queue.

Class Installation Parameters

None.

Output

Device Installation Parameters

An installer can modify the **FileQueue**, if there is one.

Return Value

A coinstaller can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error. If a class installer performs or enqueues all the necessary file operations, and thus supercedes the default handler, the class installer returns NO_ERROR.

Default Handler

SetupDiInstallDriverFiles

Operation

In response to a DIF_INSTALLDEVICEFILES request an installer specifies any necessary file operations. For example, an installer can specify an additional file to be copied that is required for device installation. If the DI_NOVCP flag is set, an installer specifies file operations by adding them to the **FileQueue** in the device installation parameters. See the Platform SDK for information on using file queues and for reference pages on file-queueing functions such as **SetupInstallFilesFromInfSection**.

If this DIF request is sent during device installation, and it completes with a Win32 error, Setup aborts the installation.

If a Setup component sends this DIF request to retrieve a list of the files associated with a device, the component retrieves the file queue but does not commit the queue.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

SetupDiInstallDriverFiles, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

DIF_INSTALLINTERFACES

A DIF_INSTALLINTERFACES request allows an installer to participate in the registration of the device interfaces for a device.

When Sent

After registering device coinstallers but before completing device installation.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Can handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData.*

Class Installation Parameters

None.

Output

Device Installation Parameters

An installer might modify the device installation parameters, but not usually for this DIF request.

Return Value

A coinstaller can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error. If a class installer installs the interfaces, including calling **SetupDiInstallDeviceInterfaces**, the class installer returns NO_ERROR.

Default Handler

SetupDiInstallDeviceInterfaces

Operation

In response to a DIF_INSTALLINTERFACES request an installer might register a device interface programmatically instead of having the interface registered through the INF file. Typically, third-party installers don't handle this DIF request.

Unless the DI_NOFILECOPY flag is set, an installer that handles this DIF request should copy files required for the device interface(s).

If the DI_NOFILECOPY flag is clear but the DI_NOVCP flag is set, the installer must enqueue any file operations to the supplied file queue but must not commit the queue.

If an installer registers a device interface, a kernel-mode component for the device (for example, a driver) must call **IoSetDeviceInterfaceState** to enable the interface.

If this DIF code completes with a Win32 error, Setup aborts the installation.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

SetupDiInstallDeviceInterfaces, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

DIF_NEWDEVICEWIZARD_FINISHINSTALL

A DIF_NEWDEVICEWIZARD_FINISHINSTALL request allows an installer to supply wizard page(s) that Setup displays to the user after the device has been installed but before Setup displays the standard finish page. Setup sends this request when installing PnP devices and manually installed nonPnP devices.

When Sent

After Setup installs a device (on successful completion of DIF_INSTALLDEVICE processing) but before it displays the Finish wizard page.

Setup sends this request during "New Hardware Found" (PnP) and "Add New Hardware" (nonPnP) device installation and during GUI-mode setup.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Can handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData*.

Class Installation Parameters

An SP_NEWDEVICEWIZARD_DATA structure is associated with the *DeviceInfoData*.

Output

Device Installation Parameters

An installer can modify the flags in the device installation parameters.

Class Installation Parameters

An installer can modify the SP_NEWDEVICEWIZARD_DATA to supply custom page(s).

Return Value

If a coinstaller does not handle this DIF request it returns NO_ERROR from its preprocessing pass. If a coinstaller handles this request it can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer returns NO_ERROR if it successfully supplies page(s). Otherwise, a class installer returns ERROR_DI_DO_DEFAULT or a Win32 error.

Default Handler

None.

Operation

A DIF_NEWDEVICEWIZARD_FINISHINSTALL request allows an installer to supply wizard page(s) that Setup displays to the user after the device has been installed but before Setup displays the standard Finish page. Setup sends this request when installing any kind of device, whether it is a PnP-enumerated device, a detected nonPnP device, or a manually installed nonPnP device.

An installer can handle this DIF request for any PnP device.

This DIF request is *not* supported when installing certain *nonPnP* devices such as modems, printers, and scanners. If a user manually installs a nonPnP printer, modem, or scanner using the Add/Remove Hardware wizard, the Setup application for that device does not display any finish-install pages supplied for the device.

An installer typically uses a custom Finish page to collect additional user preference settings, such as modem line speeds or default area codes.

Class installers and coinstallers should only supply finish-install pages if they absolutely require that the user answer questions before the device can operate properly. If the questions are optional, or if the questions were previously answered (for example, this is an upgrade for an existing device), then they should *not* supply finish-install pages. Class installers and coinstallers should always supply one or more property pages in response to DIF_ADDPROPERTYPAGE_ADVANCED so the user can adjust device settings after the device is installed.

The following pseudo-code shows how an installer adds a custom page to the array of pages in the class install parameters:

```
:
// get the class install parameters by calling
// SetupDiGetClassInstallParams

// check whether NumDynamicPages has reached the max

// fill in the PROPSHEETPAGE structure

// add the page and increment the NumDynamicPages counter
NewDevWizardData.DynamicPages[NewDevWizardData.NumDynamicPages++] =
    CreatePropertySheetPage(&Page);

// apply the modified params by calling
// SetupDiSetClassInstallParams

:
```

See the Platform SDK for information on the **CreatePropertySheetPage** function.

If an installer's custom Finish page gathers settings from the user and the device needs to be restarted to have the new settings take effect, the page dialog procedure calls **SetupDi-ChangeState** with a **StateChange** value of DICS_PROPCHANGE in the class installation parameters. This call to **SetupDiChangeState** directs the PnP Manager to query-remove and remove the device, reenumerate the device's parent, rebuild the device stack of drivers, and restart the device. If an installer determines that the device cannot be dynamically removed and restarted, the installer can set the DI_NEEDREBOOT flag in the device install parameters instead of calling **SetupDiChangeState**. An installer should not force a reboot, however, unless it is absolutely necessary.

Setup and the PnP Manager attempt to install PnP devices in a trusted PnP context without requiring user response to dialog boxes (a "server-side" installation). If an installer supplies custom Finish page(s), however, Setup is required to display those pages to the user. Setup aborts the trusted installation and restarts the device installation when a user with administrative privileges logs in (a "client-side" installation).

GUI-mode setup sends the DIF_NEWDEVICEWIZARD_FINISHINSTALL request for each device it installs, but GUI-mode setup cannot display installation wizard pages. If an installer supplies custom finish pages, GUI-mode setup marks the device so the user receives a "New Hardware Found" popup at first login. The network configuration component of GUI-mode setup locates any network devices, such as ISDN boards, that have finish-install wizard pages, gathers those pages, and incorporates them into the network setup wizard pages.

An installer should supply a Wizard 97 header title and a header subtitle in the PROPSHEETPAGE structure for a custom wizard page. An installer should not replace the system-supplied wizard title. See the Platform SDK for documentation of the PROPSHEETPAGE structure and for more information about property pages.

Installers should supply their finish-install wizard pages, regardless of the value of the DI_QUIETINSTALL flag. The sender of the DIF_NEWDEVICEWIZARD_FINISH-INSTALL request determines whether to display the wizard pages and ensures that the pages are freed in either case.

An installer can use this DIF request to add custom wizard pages, but an installer cannot replace the system-supplied finish page.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

DIF_INSTALLDEVICE, **SetupDiChangeState**, SP_DEVINFO_DATA, SP_ DEVINSTALL_PARAMS, SP_NEWDEVICEWIZARD_DATA

DIF_NEWDEVICEWIZARD_POSTANALYZE

A DIF_NEWDEVICEWIZARD_POSTANALYZE request allows an installer to supply wizard pages that Setup displays to the user after the devnode is registered but before Setup installs the drivers for the device. This request is only used during manual installation of nonPnP devices.

When Sent

After Setup registers the device, which makes the devnode "live", but before Setup installs the drivers for the device.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData.*

Class Installation Parameters

An SP_NEWDEVICEWIZARD_DATA structure is associated with the *DeviceInfoData.*

Output

Device Installation Parameters

An installer can modify the flags in the device installation parameters. Setup does not check the flags upon completion of this DIF request, but it will check them later in the installation process.

Class Installation Parameters

An installer can modify the SP_NEWDEVICEWIZARD_DATA to supply custom page(s).

Return Value

If a coinstaller does not handle this DIF request it returns NO_ERROR from its preprocessing pass. If a coinstaller handles this request it can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer returns NO_ERROR if it successfully supplies page(s). Otherwise, a class installer returns ERROR_DI_DO_DEFAULT or a Win32 error.

Default Handler

None.

Operation

A DIF_NEWDEVICEWIZARD_POSTANALYZE request allows an installer to supply wizard pages that Setup displays to the user after the devnode is registered but before Setup installs the drivers for the device. This request is only used during manual installation of nonPnP devices.

If an installer adds custom postanalyze page(s), the installer should first check whether **NumDynamicPages** in the class install parameters has reached MAX_INSTALLWIZARD_DYNAPAGES.

After the user clicks "Next" on a custom page, Setup installs the drivers for the device and the PnP Manager starts the device. A postanalyze wizard page is the last chance for an installer to do work before the drivers are loaded and the device is started.

An installer should supply a Wizard 97 header title and a header subtitle in the PROPSHEETPAGE structure for a custom wizard page. An installer should not replace the system-supplied wizard title. See the Platform SDK for documentation of the PROPSHEETPAGE structure and for more information about property pages.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

DIF_NEWDEVICEWIZARD_PREANALYZE, DIF_NEWDEVICEWIZARD_PRESELECT, DIF_NEWDEVICEWIZARD_SELECT, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS, SP_NEWDEVICEWIZARD_DATA

DIF_NEWDEVICEWIZARD_PREANALYZE

A DIF_NEWDEVICEWIZARD_PREANALYZE request allows an installer to supply wizard pages that Setup displays to the user before it displays the analyze page. This request is only used during manual installation of nonPnP devices.

When Sent

After the user has selected a driver, but before Setup registers the device which makes the devnode "live".

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData*.

Class Installation Parameters

An SP_NEWDEVICEWIZARD_DATA structure is associated with the *DeviceInfoData*.

Output

Device Installation Parameters

An installer can modify the flags in the device installation parameters. Setup does not check the flags upon completion of this DIF request, but it checks them later in the installation process.

Class Installation Parameters

An installer can modify the SP_NEWDEVICEWIZARD_DATA to supply custom wizard page(s).

Return Value

If a coinstaller does not handle this DIF request it returns NO_ERROR from its preprocessing pass. If a coinstaller handles this request it can return NO_ERROR, ERROR_DI_POST-PROCESSING_REQUIRED, or a Win32 error.

A class installer returns NO_ERROR if it successfully supplies page(s). Otherwise, a class installer returns ERROR_DI_DO_DEFAULT or a Win32 error.

Default Handler

None.

Operation

A DIF_NEWDEVICEWIZARD_PREANALYZE request allows an installer to supply wizard pages that Setup displays to the user before it displays the analyze page. These pages can be thought of as "post-select" pages. This request is only used during manual installation of nonPnP devices.

An installer might use a custom preanalyze page, for example, to choose a COM port after a modem device is selected.

If an installer adds custom preselect page(s), the installer should first check whether **NumDynamicPages** in the class install parameters has reached MAX_INSTALLWIZARD_DYNAPAGES.

An installer should supply a Wizard 97 header title and a header subtitle in the PROPSHEETPAGE structure for a custom wizard page. An installer should not replace the system-supplied wizard title. See the Platform SDK for documentation of the PROPSHEETPAGE structure and for more information about property pages.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

DIF_NEWDEVICEWIZARD_PRESELECT, DIF_NEWDEVICEWIZARD_POSTANALYZE, DIF_NEWDEVICEWIZARD_SELECT, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS, SP_NEWDEVICEWIZARD_DATA

DIF_NEWDEVICEWIZARD_PRESELECT

A DIF_NEWDEVICEWIZARD_PRESELECT request allows an installer to supply wizard pages that Setup displays to the user before it displays the select-driver page. This request is only used during manual installation of nonPnP devices.

When Sent

After the user has selected the class for the device but before Setup displays the "Select a Device Driver" page.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData*.

Class Installation Parameters

An SP_NEWDEVICEWIZARD_DATA structure is associated with the *DeviceInfoData*.

Output

Device Installation Parameters

An installer can modify the flags in the device installation parameters. Setup does not check the flags upon completion of this DIF request, but it checks them later in the installation process.

Class Installation Parameters

An installer can modify the SP_NEWDEVICEWIZARD_DATA to supply custom page(s).

Return Value

If a coinstaller does not handle this DIF request it returns NO_ERROR from its preprocessing pass. If a coinstaller handles this request it can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer returns NO_ERROR if it successfully supplies page(s). Otherwise, a class installer returns ERROR_DI_DO_DEFAULT or a Win32 error.

Default Handler

None.

Operation

A DIF_NEWDEVICEWIZARD_PRESELECT request allows an installer to supply wizard pages that Setup displays to the user before it displays the select-driver page. This request is only used during manual installation of nonPnP devices.

If an installer adds custom preselect page(s), the installer should first check whether **NumDynamicPages** in the class install parameters has reached MAX_INSTALLWIZARD_DYNAPAGES.

A coinstaller can add custom pages in its preprocessing pass and/or in its postprocessing pass. If it adds page(s) in its preprocessing pass, those pages are displayed before any page(s) supplied by the class installer.

If one or more installers add custom preselect pages, Setup displays the pages before the "Select a Device Driver" page. However, if the user presses "Back" on the select-driver page, Setup skips the custom preselect pages and goes back to the "Hardware Type" class-selection page.

An installer should supply a Wizard 97 header title and a header subtitle in the PROPSHEETPAGE structure for a custom wizard page. An installer should not replace the system-supplied wizard title. See the Platform SDK for documentation of the PROPSHEETPAGE structure and for more information about property pages.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

DIF_NEWDEVICEWIZARD_PREANALYZE, DIF_NEWDEVICEWIZARD_POSTANALYZE, DIF_NEWDEVICEWIZARD_SELECT, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS, SP_NEWDEVICEWIZARD_DATA

DIF_NEWDEVICEWIZARD_SELECT

A DIF_NEWDEVICEWIZARD_SELECT request allows an installer to supply custom wizard page(s) that replace the standard select-driver page. This request is only used during manual installation of nonPnP devices.

When Sent

Immediately before Setup displays the "Select a Device Driver" page.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData*.

Class Installation Parameters

An SP_NEWDEVICEWIZARD_DATA structure is associated with the *DeviceInfoData*.

Output

Device Installation Parameters

An installer can modify the flags in the device installation parameters. Setup does not check the flags upon completion of this DIF request, but it checks them later in the installation process.

Class Installation Parameters

An installer can modify the SP_NEWDEVICEWIZARD_DATA to supply custom page(s).

Return Value

If a coinstaller does not handle this DIF request it returns NO_ERROR from its preprocessing pass. If a coinstaller handles this request it can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer returns NO_ERROR if it successfully supplies page(s). Otherwise, a class installer returns ERROR_DI_DO_DEFAULT or a Win32 error.

Default Handler

None.

Operation

A DIF_NEWDEVICEWIZARD_SELECT request allows an installer to supply custom wizard page(s) that replace the standard select-driver page. This request is only used during manual installation of nonPnP devices.

An installer responds to this DIF request to completely replace the standard select-driver wizard page. If, instead, the installer only needs to modify the standard page or modify the list of drivers from which to choose, the installer should do so in response to the DIF_SELECTDEVICE request.

A coinstaller should add custom page(s) in its postprocessing pass and only if the class installer did not add custom page(s). If the class installer added page(s), the coinstaller should not. Otherwise, the user might be asked to choose a driver twice.

If an installer supplies a custom select page, the installer must set the selected driver. In the installer's code that supports the wizard page, after the user clicks "Next", the installer must call **SetupDiSetSelectedDriver**.

An installer should supply a Wizard 97 header title and a header subtitle in the PROPSHEETPAGE structure for a custom wizard page. An installer should not replace the system-supplied wizard title. See the Platform SDK for documentation of the PROPSHEETPAGE structure and for more information about property pages.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

DIF_NEWDEVICEWIZARD_PREANALYZE, DIF_NEWDEVICEWIZARD_PRESELECT, DIF_NEWDEVICEWIZARD_POSTANALYZE, DIF_SELECTDEVICE, **SetupDiSetSelectedDevice**, **SetupDiSetSelectedDriver**, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS, SP_NEWDEVICEWIZARD_DATA

DIF_POWERMESSAGEWAKE

A DIF_POWERMESSAGEWAKE request allows an installer to supply custom text that Setup displays on the power management properties page of the device properties.

When Sent

When a user clicks on a menu item or tab to display the properties of a device.

Setup only sends this DIF request if the drivers for the device support power management. Otherwise, Setup does not display any power properties for the device.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Can handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData*.

Class Installation Parameters

An SP_POWERMESSAGEWAKE_PARAMS structure is associated with the *DeviceInfoData*.

Output

Class Installation Parameters

An installer can modify the SP_POWERMESSAGEWAKE_PARAMS to supply custom text for a device's power properties page.

Return Value

A coinstaller typically returns NO_ERROR, ERROR_DI_POSTPROCESSING_ REQUIRED, or a Win32 error.

A class installer returns NO_ERROR if it successfully supplies power properties text. Otherwise, a class installer returns ERROR_DI_DO_DEFAULT or a Win32 error.

Default Handler

None.

Operation

A DIF_POWERMESSAGEWAKE request allows an installer to supply text that Setup displays on the power properties page for a device.

If a coinstaller supplies power-properties text, it should do so in its postprocessing phase. A coinstaller should take care when overwriting any power-properties text supplied by an installer that handled the request before the coinstaller.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS, SP_POWERMESSAGEWAKE_ PARAMS

DIF_PROPERTYCHANGE

A DIF_PROPERTYCHANGE request notifies the installer that the device's properties are changing. The device is being enabled, disabled, started, stopped, or some item on a property page has changed. This DIF request gives the installer an opportunity to participate in the change.

When Sent

When a device is being enabled, disabled, started, stopped, or its properties have changed.

For example, Setup sends this request when a property-page provider sets the DI_FLAGSEX_PROPCHANGE_PENDING flag in the **FlagsEx** field of the SP_DEVINSTALL_PARAMS structure for the device.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Can handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure for the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData*.

Class Installation Parameters

An SP_PROPCHANGE_PARAMS structure is associated with the *DeviceInfoData*.

Output

None.

Return Value

A coinstaller can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error. If a class installer fully handles the property change request, including calling or superceding the default handler, the class installer returns NO_ERROR.

Default Handler

SetupDiChangeState

Operation

In response to a DIF_PROPERTYCHANGE request an installer can participate in the property-change operation. The class installation parameters (SP_PROPCHANGE_PARAMS) indicate which change is taking place.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

SetupDiChangeState, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS, SP_PROPCHANGE_PARAMS

DIF_REGISTER_COINSTALLERS

A DIF_REGISTER_COINSTALLERS request allows an installer to participate in the registration of device-specific coinstallers for a device.

When Sent

Before completing device installation.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device for which coinstallers are to be registered.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData*.

Class Installation Parameters

None.

Output

None.

Return Value

A coinstaller can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error. If a class installer fully handles the registration request, including calling or superceding the default handler, the class installer returns NO_ERROR.

Default Handler

SetupDiRegisterCoDeviceInstallers

Operation

In response to a DIF_REGISTER_COINSTALLERS request an installer might modify the list of coinstallers for the device. For example, an installer might programmatically register or remove a device-specific coinstaller for the device based on analysis of the device.

Unless the DI_NOFILECOPY flag is set, an installer that handles this DIF request should copy files required for the coinstaller(s).

If the DI_NOFILECOPY flag is clear but the DI_NOVCP flag is set, the installer must enqueue any file operations to the supplied file queue but must not commit the queue.

If this DIF code completes with a Win32 error, Setup aborts the installation.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

SetupDiRegisterCoDeviceInstallers, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

DIF_REGISTERDEVICE

The DIF_REGISTERDEVICE request allows an installer to participate in registering a newly created device instance with the PnP Manager. Setup sends this DIF request for nonPnP devices.

When Sent

When an installer reports a previously unknown device in response to a DIF_DETECT request or if a user manually selects a device in the Add/Remove Hardware wizard. Setup sends this DIF request in the analyze phase of the Add Hardware Wizard before it installs the device. Setup also sends this request during nonPnP detection.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData.*

Class Installation Parameters

None.

Output

None.

Return Value

A coinstaller can return NO_ERROR or a Win32 error. A coinstaller should not return ERROR_DI_POSTPROCESSING_REQUIRED for this DIF request.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error. If a class installer completely registers the device, including performing duplicate detection, the class installer returns NO_ERROR.

If an installer determines that the device is a duplicate it returns ERROR_DUPLICATE_ FOUND.

Default Handler

SetupDiRegisterDeviceInfo

Operation

A setup application typically sends this DIF request to register a nonPnP device with the PnP Manager. NonPnP devices must be registered before they can be installed on Windows 2000.

An installer typically handles this DIF request to do duplicate detection. Such an installer typically calls the default handler (**SetupDiRegisterDeviceInfo**) and specifies its detection routine. If the registration is successful and the installer determines that the device is not a duplicate, the installer returns NO_ERROR.

A coinstaller should perform any operations to handle this DIF request in its preprocessing pass. When the coinstaller is called for postprocessing, the device instance has already been registered by either the class installer or the default handler.

If an installer returns an error for this DIF code, typically ERROR_DUPLICATE_FOUND, Setup deletes the device from the device information set.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

DIF_DETECT, **SetupDiRegisterDeviceInfo**, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

DIF_REMOVE

A DIF_REMOVE request notifies an installer that Setup is about to remove a device and gives the installer an opportunity to prepare for the removal.

When Sent

When a user removes a device in the Device Manager or in the Add/Remove Hardware wizard.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Can handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device to be removed.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure for the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData.*

Class Installation Parameters

An SP_REMOVEDEVICE_PARAMS structure might be associated with the *Device-InfoData.*

There are no class installation parameters for the request if the DI_CLASSINSTALL-PARAMS flag is clear in the SP_DEVINSTALL_PARAMS. In this case, no hardware profile is specified and the device is to be removed from the system as a whole.

Output

None.

Return Value

A coinstaller can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error. If a class installer fully handles this request, including calling or superceding the default handler, the class installer returns NO_ERROR.

Default Handler

SetupDiRemoveDevice

Operation

In response to a DIF_REMOVE request, an installer typically performs some clean-up operations. In this case, a coinstaller returns NO_ERROR and a class installer returns ERROR_DI_DO_DEFAULT.

If an installer determines that the device should not be removed, the installer fails the DIF request by returning a Win32 error. If the DI_QUIETINSTALL flag is clear, the installer should display a message to the user explaining why the device is not being removed.

Coinstallers must not attempt to remove the device themselves by calling **SetupDiRemove-Device**. Coinstallers typically handle this request in postprocessing, after the device has been successfully removed.

If a coinstaller needs to delete information in the registry, for example, the coinstaller should do so in postprocessing and only if the previous installers succeeded the removal request. In its preprocessing pass, the coinstaller should store the registry information in its context parameter and return ERROR_DI_POSTPROCESSING_REQUIRED to request postprocessing. When Setup calls the coinstaller for postprocessing of this DIF request, the coinstaller should check that the DIF status is NO_ERROR and then delete the registry information. If a coinstaller deletes registry information in its preprocessing pass and the class installer (or another coinstaller) fails the DIF_REMOVE, the coinstaller could leave the device in an unpredictable state.

Installers should not delete files when handling this DIF request, in case the files are in use by another device.

Setup sends this DIF request before it initiates PnP query-remove and remove processing.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

SetupDiRemoveDevice, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS, SP_REMOVEDEVICE_PARAMS

DIF_SELECTBESTCOMPATDRV

A DIF_SELECTBESTCOMPATDRV request allows an installer to select the best driver from the device information element's compatible driver list.

When Sent

When the OS is preparing to install a new PnP device or is performing a change-driver operation on a PnP device.

This DIF request is typically used during a PnP configuration. If a device is being manually installed, Setup sends a DIF_SELECTDEVICE request.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData.*

Class Installation Parameters

None.

Output

Device Installation Parameters

An installer can modify the device installation parameters, but they typically do not when handling this DIF request.

DeviceInfoData

As a side effect, an installer can modify the driver list associated with the *DeviceInfoData*, in particular, the SP_DRVINSTALL_PARAMS.

Return Value

A coinstaller can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error. In some cases, a class installer returns NO_ERROR.

Default Handler

SetupDiSelectBestCompatDrv

Operation

An installer handles this DIF request to participate in selecting a driver for a PnP device. An installer typically responds to this DIF request in one of the following ways:

- Do nothing.

  If an installer has no special selection requirements, it does nothing in response to this DIF request. A class installer returns ERROR_DI_DO_DEFAULT and a coinstaller returns NO_ERROR.

- Modify the parameters of one or more drivers in the driver list.

For example, an installer might remove a driver from consideration for the device by marking it DNF_BAD_DRIVER. An installer modifies driver parameters with a procedure like the following:

1. Get the information about the first driver in the list by calling **SetupDiEnumDriverInfo** and **SetupDiGetDriverInstallParams**. If appropriate, modify the driver parameters and apply the change by calling **SetupDiSetDriverInstallParams**.

   If a driver is a worst-case choice, set the driver's rank to 0xFFFF or higher in the driver install parameters. See *How Does Setup Select a Driver For a Device?* in Part 4, "Setup," in the *Plug and Play, Power Management, and Setup Design Guide*for more information.

2. Repeat the previous step until you have processed all the drivers in the list. Be sure to increment the *MemberIndex* parameter to **SetupDiEnumDriverInfo** as described in the reference page for that function.

   After a class installer modifies the driver list, it returns ERROR_DI_DO_DEFAULT. If a coinstaller modifies the driver list, it should do so in preprocessing and return NO_ERROR.

- Select the best driver for the device.

  This action is less common, but an installer might choose the best driver for the device. Such an installer would examine the data for each driver, choose a driver, and call **SetupDiSetSelectedDriver** to set the driver. After an installer sets the selected driver, it returns NO_ERROR.

  If a coinstaller selects a driver, it should do so in postprocessing.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion of handling DIF codes in an installer.

See Also

SetupDiSelectBestCompatDrv, **SetupDiSetSelectedDriver**, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

DIF_SELECTDEVICE

A DIF_SELECTDEVICE request allows an installer to participate in selecting the driver for a device.

When Sent

When choosing a driver for a newly enumerated device or a new driver for an existing device (change driver). For example, when a user selects Add/Remove Hardware and selects the modem class. Or, a user inserts a PnP device and selects "Choose a Driver From a List" in the Found New Hardware Wizard.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device for which a driver is to be selected.

Associated Class?

There is a device setup class associated with the *DeviceInfoSet*.

DeviceInfoData

Optionally supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

If *DeviceInfoData* is NULL, this request is to select a driver for the device setup class associated with the *DeviceInfoSet*.

Device Installation Parameters

If *DeviceInfoData* is not NULL, there are device installation parameters (SP_ DEVINSTALL_PARAMS) associated with the *DeviceInfoData*. If *DeviceInfoData* is NULL, there are device installation parameters associated with the *DeviceInfoSet*.

Of particular interest is the **DriverPath**, which contains the location of INF(s) to use when building the driver list.

Class Installation Parameters

An SP_SELECTDEVICE_PARAMS structure is associated with the *DeviceInfoData* if *DeviceInfoData* is not NULL. Otherwise, the class installation parameters are associated with the device information set as a whole.

Output

Device Installation Parameters

An installer can modify the device installation parameters, but it should not modify the **DriverPath** field.

Class Installation Parameters

An installer can modify the SP_SELECTDEVICE_PARAMS. For example, an installer might specify a title and/or instructions for Setup to use in the dialog box that asks the user to select a driver.

If an installer sets new select-device parameters, vs. modifying parameters set by a previous installer, the installer must zero the fields it does not set.

Return Value

If a coinstaller does nothing for this DIF code, it returns NO_ERROR from its preprocessing pass. If a coinstaller handles this DIF code, it should do so in its preprocessing pass and return NO_ERROR or a Win32 error. By the time a coinstaller is called for postprocessing, the driver has already been selected.

A class installer returns ERROR_DI_DO_DEFAULT, a Win32 error, or NO_ERROR.

If an installer builds and modifies the driver list, it returns ERROR_DI_BAD_PATH if SP_DEVINSTALL_PARAMS.**DriverPath** is not NULL yet there are no valid drivers at that location. This can be true if there are no drivers at that location or if there are drivers, but only ones that the installer marks DNF_BAD_DRIVER. In response to this error code, Setup displays an error to the user.

Default Handler

SetupDiSelectDevice

Operation

In response to a DIF_SELECTDEVICE request, an installer performs any selection operations required for its device or device class, besides what the default handler does. An installer typically responds to this DIF request in one of the following ways:

- Do nothing.

  If an installer has no special selection requirements, it does nothing in response to this DIF code. A class installer returns ERROR_DI_DO_DEFAULT and a coinstaller returns NO_ERROR.

- Supply select strings that Setup will display in the selection UI.

  An installer can supply select strings in the class installation parameters (SP_SELECT-DEVICE_PARAMS). For example, an installer can modify the **Instructions** or the window header **Title**.

  A class installer should not supply select strings if a coinstaller already supplied select strings. The coinstaller probably has more relevant information.

If an installer modifies the SP_SELECTDEVICE_PARAMS, the installer must also set the DI_USECI_SELECTSTRINGS flag in the SP_DEVINSTALL_PARAMS.

If an installer successfully supplies select strings, Setup still needs to call the default handler. Therefore, in this case, a coinstaller returns NO_ERROR and a class installer returns ERROR_DI_DO_DEFAULT.

- Modify the device installation parameters.

  An installer can modify the device installation parameters (SP_DEVINSTALL_PARAMS). For example, an installer might set the DI_SHOWOEM flag to have Setup display the Have Disk button.

  If a class installer successfully modifies the device installation parameters, the class installer returns ERROR_DI_DO_DEFAULT.

- Modify the list of drivers from which the user can select.

  This action is less common, but possible. An installer that modifies the driver list might, or might not, also supply select strings.

  An installer that modifies the driver list typically marks driver(s) that are inappropriate for the device. An installer marks such drivers with the flag DNF_BAD_DRIVER. Setup omits these drivers from the list it displays to the user. An installer marks bad drivers with a procedure like the following:

  1. Build the driver list by calling **SetupDiBuildDriverInfoList** with a *DriverType* of SPDIT_CLASSDRIVER.
  2. Get the information about the first driver in the list by calling **SetupDiEnumDriverInfo** and **SetupDiGetDriverInstallParams**. If the driver is not appropriate for the device, set the DNF_BAD_DRIVER flag in the **Flags** field of the parameters. Apply the change to the parameters by calling **SetupDiSetDriverInstallParams**.
  3. Repeat the previous step until you have processed all the drivers in the list. Be sure to increment the *MemberIndex* parameter to **SetupDiEnumDriverInfo** as described in the reference page for that function.

  An installer might set the DNF_BAD_DRIVER flag for one or more drivers in the driver list, but an installer must not clear that flag.

  If one or more installers successfully modify the driver list, Setup still needs to call the default handler. Therefore, in this case, a coinstaller returns NO_ERROR and a class installer returns ERROR_DI_DO_DEFAULT.

- Display its own driver-selection user interface and set the selected driver.

Only a class installer can display its own driver-selection user interface; coinstallers must not. For example, a class installer might display pictures instead of textual lists.

If the class installer successfully sets the selected driver, the class installer returns NO_ERROR and Setup does not call the default handler and therefore does not display the default selection interface.

If a class installer only needs to display its own interface during the Add New Hardware wizard, the class installer should handle the DIF_NEWDEVICEWIZARD_SELECT request instead of this request.

If the DI_ENUMSINGLEINF flag is set in the device installation parameters, the **Driver-Path** is a path to a single INF instead of a path to a directory. An installer must use only that single INF to build the driver list.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

DIF_NEWDEVICEWIZARD_SELECT, **SetupDiSelectDevice**, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS, SP_SELECTDEVICE_PARAMS

DIF_TROUBLESHOOTER

The DIF_TROUBLESHOOTER request allows an installer to launch a troubleshooter for a device or to return CHM and HTM troubleshooter files for Setup to launch.

When Sent

When a user clicks the "Troubleshooter" button for a device in the Device Manager.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Can handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData*.

Class Installation Parameters

An SP_TROUBLESHOOTER_PARAMS structure is associated with the *DeviceInfoData*.

Output

Class Installation Parameters

An installer might modify the SP_TROUBLESHOOTER_PARAMS, setting a CHM or HTML file.

Return Value

If a coinstaller does not handle this request, it returns NO_ERROR from its preprocessing pass.

If a coinstaller handles this request, it does so in its postprocessing pass. If the coinstaller supplies CHM and HTML files, it propagates the status it received (probably ERROR_DI_DO_DEFAULT). If the coinstaller runs a troubleshooter and fixes the problem, the coinstaller returns NO_ERROR. If the coinstaller runs a troubleshooter but does not fix the problem, it propagates the status it received (ERROR_DI_DO_DEFAULT).

If a class installer launches its own troubleshooter and fixes the problem, the class installer returns NO_ERROR. If a class installer supplies a CHM file and an HTML file or the class installer runs a troubleshooter but does not fix the problem, the class installer returns ERROR_DI_DO_DEFAULT.

If an installer encounters an error when handling this DIF code, the installer returns an appropriate Win32 error.

Default Handler

None.

There is no default handler for DIF_TROUBLESHOOTER, but the OS provides default troubleshooters that attempt to resolve device problems if there are no installer-supplied troubleshooters.

Operation

An installer calls **CM_Get_DevNode_Status** to get the device status and the CM problem code. Depending on the problem, an installer might provide a troubleshooter, a help file, or nothing. A troubleshooter can possibly resolve a problem with a device. If a troubleshooter resolves the problem, it should call **SetupDiCallClassInstaller** to send a DIF_PROPERTYCHANGE request of type DICS_PROPCHANGE. If an installer does not supply a troubleshooter for a device, it might supply a help file of problem-solving suggestions for the user.

If no installer runs its own troubleshooter, Setup runs HTML Help to display information to the user. If an installer supplied a CHM file in the class installation parameters, Setup displays that file. Otherwise, Setup displays system-supplied troubleshooting information.

The class installation parameters contain at most one **ChmFile** and **HtmlTroubleshooter** pair. If more than one installer specifies these values, Setup uses the values set by the last installer that handled the DIF request.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

CM_Get_DevNode_Status, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS, SP_TROUBLESHOOTER_PARAMS

DIF_UNREMOVE

A DIF_UNREMOVE request notifies the installer that Setup is about to reinstate a device in a given hardware profile and gives the installer an opportunity to participate in the operation. Setup only sends this request for nonPnP devices.

When Sent

When a root-enumerated, nonPnP device is reinstated to a hardware profile.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Can handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set containing the device.

DeviceInfoData

Supplies a pointer to an SP_DEVINFO_DATA structure that identifies the device in the device information set.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoData.*

Class Installation Parameters

An SP_UNREMOVEDEVICE_PARAMS structure is associated with the *DeviceInfoData.* The **Scope** field must be set to DI_UNREMOVEDEVICE_CONFIGSPECIFIC and a hardware profile must be specified in the **HwProfile** field.

Output

None.

Return Value

A coinstaller can return NO_ERROR, ERROR_DI_POSTPROCESSING_REQUIRED, or a Win32 error.

A class installer typically returns ERROR_DI_DO_DEFAULT or a Win32 error. In some cases, a class installer returns NO_ERROR.

Default Handler

SetupDiUnRemoveDevice

Operation

"Unremoving" a device essentially means that Setup clears a flag that previously marked a device as "not present" in a particular hardware profile.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

SetupDiUnRemoveDevice, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS, SP_UNREMOVEDEVICE_PARAMS

Reserved DIF Codes

The DIF codes listed in this section are reserved for system use. Third-party installers should not handle or send these requests.

DIF_ADDPROPERTYPAGE_BASIC

This DIF request is reserved for system use. Third-party installers must not handle this request.

DIF_ASSIGNRESOURCES

This DIF request is reserved for system use. Third-party installers must not handle this request.

DIF_CALCDISKSPACE

This DIF request is reserved for system use. Third-party installers must not handle this request.

DIF_DETECTCANCEL

This DIF request is reserved for system use. Third-party installers must not handle this request.

DIF_DETECTVERIFY

This DIF request is reserved for system use. Third-party installers must not handle this request.

DIF_ENABLECLASS

This DIF request is reserved for system use. Third-party installers must not handle this request.

DIF_FIRSTTIMESETUP

A DIF_FIRSTTIMESETUP request directs an installer to carry out any class-specific installation tasks that need to be completed during the initial installation of the operating system.

This DIF request is reserved. Only system-supplied installers are allowed to handle this DIF request.

When Sent

During GUI-mode setup.

Who Handles

Class Coinstaller	Can handle
Device Coinstaller	Does not handle
Class Installer	Can handle

Input

DeviceInfoSet

Supplies a handle to the device information set.

Associated Class?

There is a device setup class associated with the *DeviceInfoSet.*

DeviceInfoData

None.

Device Installation Parameters

There are device installation parameters (SP_DEVINSTALL_PARAMS) associated with the *DeviceInfoSet.*

Class Installation Parameters

None.

Output

DeviceInfoSet

An installer adds a device information element to the *DeviceInfoSet* for each detected device it wants to have installed. An installer might also build a global class driver list.

Device Installation Parameters

An installer can modify the device installation parameters for the *DeviceInfoSet* or for new device information elements it creates.

Return Value

A class coinstaller can detect devices during pre- or postprocessing. Such a coinstaller returns ERROR_DI_POSTPROCESSING_REQUIRED (for postprocessing) and/or returns NO_ERROR or a Win32 error after its detection operations. If a coinstaller does not detect devices, it returns NO_ERROR from its preprocessing pass.

If a class installer detects devices, it returns NO_ERROR or an appropriate Win32 error. If a class installer does not handle this DIF request, it returns ERROR_DI_DO_DEFAULT.

Default Handler

None.

Operation

GUI-mode setup sends a DIF_FIRSTTIMESETUP request with an empty *DeviceInfoSet*. The installers can perform legacy detection of nonPnP devices and add them to the *Device-InfoSet*. System-supplied installers can also handle this DIF request when migrating legacy device installations from Windows 9x or Windows NT® to Windows 2000.

An installer detects new devices of its setup class, based on registry information, by calling into a kernel-mode detection component, or by consulting *unattend.txt* information stored when a migration DLL ran during an OS upgrade.

If an installer detects a nonPnP device, the installer should select a driver for the device as follows: create a device information element (**SetupDiCreateDeviceInfo**), set the SPDRP_HARDWAREID property by calling **SetupDiSetDeviceRegistryProperty**, call **SetupDiBuildDriverInfoList**, and then call **SetupDiCallClassInstaller** to send a DIF_SELECTBESTCOMPATDRV request.

If one or more installers detect device(s) in response to this DIF code, GUI-mode setup attempts to install the device(s). GUI-mode setup attempts to install all devices in the list; if an installer returns a device that was previously configured, GUI-mode setup will install the device twice.

An installer must handle this DIF request silently. That is, without displaying UI to the user.

Installers should not perform tasks when handling this DIF request that require the machine to be rebooted. For example, a class installer should not set drivers to load at the next boot for the purpose of determining which drivers succeed after the reboot.

To detect nonPnP devices during GUI-mode setup, an installer must handle this request. GUI-mode setup does not send a DIF_DETECT request.

See the *Plug and Play, Power Management, and Setup Design Guide* for an overall discussion on handling DIF codes in an installer.

See Also

DIF_SELECTBESTCOMPATDRV, **SetupDiBuildDriverInfoList**, **SetupDiCall-ClassInstaller**, **SetupDiCreateDeviceInfo**, **SetupDiSetDeviceRegistryProperty**, SP_DEVINFO_DATA, SP_DEVINSTALL_PARAMS

DIF_FOUNDDEVICE

This DIF request is reserved for system use. Third-party installers must not handle this request.

DIF_INSTALLCLASSDRIVERS

This DIF request is reserved for system use. Third-party installers must not handle this request.

DIF_MOVEDEVICE

This DIF request is reserved for system use. Third-party installers must not handle this request.

Obsolete DIF Codes

The DIF codes listed in this section are obsolete. Third-party installers should not handle or send these requests.

DIF_DESTROYWIZARDDATA

This DIF code is obsolete.

Setup uses the DIF_NEWDEVICEWIZARD_*XXX* requests instead, such as DIF_NEWDEVICEWIZARD_FINISHINSTALL.

DIF_INSTALLWIZARD

This DIF code is obsolete.

For PnP devices, Setup uses the DIF_NEWDEVICEWIZARD_*XXX* requests instead, such as DIF_NEWDEVICEWIZARD_FINISHINSTALL.

DIF_PROPERTIES

This DIF code is obsolete.

To supply custom property pages for a device an installer handles the DIF_ADDPROPERTYPAGE_ADVANCED request.

DIF_SELECTCLASSDRIVERS

This DIF request is obsolete.

DIF_VALIDATECLASSDRIVERS

This DIF code is obsolete.

DIF_VALIDATEDRIVER

This DIF code is obsolete.

C H A P T E R 6

PnP Configuration Manager Functions

This section describes PnP Configuration Manager functions, which can be called by class installers, co-installers, and additional installation applications running under Microsoft® Windows NT®/Windows® 2000. These functions are typically used in conjunction with device installation functions. Function prototypes are defined in *cfgmgr32.h.* Additional definitions are in *cfg.h.*

CM_Add_Empty_Log_Conf

```
CMAPI CONFIGRET WINAPI
  CM_Add_Empty_Log_Conf(
    OUT PLOG_CONF plcLogConf,
    IN DEVINST dnDevInst,
    IN PRIORITY Priority,
    IN ULONG ulFlags
    );
```

The **CM_Add_Empty_Log_Conf** function creates an empty logical configuration, for a specified configuration type and a specified device instance, on the local system.

Parameters

plcLogConf

Address of a location to receive the handle to an empty logical configuration.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex

CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

Priority

Caller-supplied configuration priority value. This must be one of the constant values listed in the following table. The constants are listed in order of priority, from highest to lowest. (For multiple configurations with the same *ulFlags* value, the system will attempt to use the one with the highest priority first.)

Priority Constant	Definition
LCPRI_FORCECONFIG	Result of a forced configuration.
LCPRI_BOOTCONFIG	Result of a boot configuration.
LCPRI_DESIRED	Preferred configuration (better performance).
LCPRI_NORMAL	Workable configuration (acceptable performance).
LCPRI_LASTBESTCONFIG	*For internal use only.*
LCPRI_SUBOPTIMAL	Not a desirable configuration, but it will work.
LCPRI_LASTSOFTCONFIG	*For internal use only.*
LCPRI_RESTART	The system must be restarted
LCPRI_REBOOT	The system must be restarted (same as LCPRI_RESTART).
LCPRI_POWEROFF	The system must be shut down and powered off.
LCPRI_HARDRECONFIG	A jumper must be changed.
LCPRI_HARDWIRED	The configuration cannot be changed.
LCPRI_IMPOSSIBLE	The configuration cannot exist.
LCPRI_DISABLED	Disabled configuration.

ulFlags

Caller-supplied flags that specify the type of the logical configuration. One of the following flags must be specified:

Configuration Type Flags	Definitions
BASIC_LOG_CONF	Resource descriptors added to this configuration will describe a basic configuration.
FILTERED_LOG_CONF	*Do not use.* (Only the PnP manager can create a filtered configuration.)

Configuration Type Flags	Definitions
ALLOC_LOG_CONF	*Do not use.* (Only the PnP manager can create an allocated configuration.)
BOOT_LOG_CONF	Resource descriptors added to this configuration will describe a boot configuration.
FORCED_LOG_CONF	Resource descriptors added to this configuration will describe a forced configuration.
OVERRIDE_LOG_CONF	Resource descriptors added to this configuration will describe an override configuration.

One of the following bit flags can be OR'ed with the configuration type flag.

Priority Comparison Flags	Definitions
PRIORITY_EQUAL_FIRST	If multiple configurations of the same type (*ulFlags*) have the same priority (*Priority*), this configuration is placed at the head of the list.
PRIORITY_EQUAL_LAST	(Default) If multiple configurations of the same type (*ulFlags*) have the same priority (*Priority*), this configuration is placed at the tail of the list.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

Calling **CM_Add_Empty_Log_Conf** can cause the handles returned by **CM_Get_First_Log_Conf** and **CM_Get_Next_Log_Conf** to become invalid. Thus if you want to obtain logical configurations after calling **CM_Add_Empty_Log_Conf**, your code must call **CM_Get_First_Log_Conf** again and start at the first configuration.

To remove a logical configuration created by **CM_Add_Empty_Log_Conf**, call **CM_Free_Log_Conf**.

The handle received in *plcLogConf* must be explicitly freed by calling **CM_Free_Log_Conf_Handle**.

See Also

CM_Add_Empty_Log_Conf_Ex

CM_Add_Empty_Log_Conf_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Add_Empty_Log_Conf_Ex(
    OUT PLOG_CONF plcLogConf,
    IN DEVINST dnDevInst,
    IN PRIORITY Priority,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Add_Empty_Log_Conf_Ex** function creates an empty logical configuration, for a specified configuration type and a specified device instance, on either the local or a remote system.

Parameters

plcLogConf

Pointer to a location to receive the handle to an empty logical configuration.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

Priority

Caller-supplied configuration priority value. For a list of values, see the *Priority* description for **CM_Add_Empty_Log_Conf**.

ulFlags

Caller-supplied flags that specify the type of the logical configuration. For a list of flags, see the description *ulFlags* description for **CM_Add_Empty_Log_Conf**.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**, or NULL.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

Calling **CM_Add_Empty_Log_Conf_Ex** can cause the handles returned by **CM_Get_First_Log_Conf_Ex** and **CM_Get_Next_Log_Conf_Ex** to become invalid. Thus if you want to obtain logical configurations after calling **CM_Add_Empty_Log_Conf_Ex**, your code must call **CM_Get_First_Log_Conf_Ex** again and start at the first configuration.

To remove a logical configuration created by **CM_Add_Empty_Log_Conf_Ex**, call **CM_Free_Log_Conf_Ex**.

The handle received in *plcLogConf* must be explicitly freed by calling **CM_Free_Log_Conf_Handle**.

See Also

CM_Add_Empty_Log_Conf

CM_Add_ID

```
CMAPI CONFIGRET WINAPI
  CM_Add_ID(
    IN DEVINST dnDevInst,
    IN PTSTR pszID,
    IN ULONG ulFlags
    );
```

The **CM_Add_ID** function appends a specified device ID (if not already present) to a device instance's hardware ID list or compatible ID list.

Parameters

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

pszID

Caller-supplied pointer to a NULL-terminated device ID string.

ulFlags

Caller-supplied flag constant indicating the list onto which the supplied device ID should be appended. The following flag constants are valid.

Flag Constant	Definition
CM_ADD_ID_COMPATIBLE	The specified device ID should be appended to the specific device instance's compatible ID list.
CM_ADD_ID_HARDWARE	The specified device ID should be appended to the specific device instance's hardware ID list.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

Each appended device ID is considered less compatible than IDs already existing in the specified list.

See Also

CM_Add_ID_Ex SetupDiSetDeviceRegistryProperty

CM_Add_ID_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Add_ID(
    IN DEVINST dnDevInst,
    IN PTSTR pszID,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Add_ID_Ex** function appends a device ID (if not already present) to a device instance's hardware ID list or compatible ID list, on either the local or a remote system.

Parameters

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

pszID

Caller-supplied pointer to a NULL-terminated device ID string.

ulFlags

Caller-supplied flag constant indicating the list onto which the supplied device ID should be appended. The following flag constants are valid.

Flag Constant	**Definition**
CM_ADD_ID_COMPATIBLE	The specified device ID should be appended to the specific device instance's compatible ID list.
CM_ADD_ID_HARDWARE	The specified device ID should be appended to the specific device instance's hardware ID list.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**, or NULL.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

Each appended device ID is considered less compatible than IDs already existing in the specified list.

See Also

CM_Add_ID SetupDiSetDeviceRegistryProperty

CM_Add_Res_Des

```
CMAPI CONFIGRET WINAPI
  CM_Add_Res_Des(
    OUT PRES_DES prdResDes,
    IN LOG_CONF lcLogConf,
    IN RESOURCEID ResourceID,
    IN PCVOID ResourceData,
    IN ULONG ResourceLen,
    IN ULONG ulFlags
    );
```

The **CM_Add_Res_Des** function adds a resource descriptor to a logical configuration.

Parameters

prdResDes

Pointer to a location to receive a handle to the new resource descriptor.

lcLogConf

Caller-supplied handle to the logical configuration to which the resource descriptor should be added. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Empty_Log_Conf
CM_Add_Empty_Log_Conf_Ex
CM_Get_First_Log_Conf
CM_Get_First_Log_Conf_Ex
CM_Get_Next_Log_Conf
CM_Get_Next_Log_Conf_Ex

ResourceID

Caller-supplied resource type identifier, which identifies the type of structure supplied by *ResourceData*. This must be one of the **ResType_**-prefixed constants defined in *cfgmgr32.h*.

ResourceData

Caller-supplied pointer to one of the resource structures listed in the following table.

ResourceID Parameter	Resource Structure
ResType_BusNumber	BUSNUMBER_RESOURCE
ResType_ClassSpecific	CS_RESOURCE
ResType_DevicePrivate	DEVPRIVATE_RESOURCE
ResType_DMA	DMA_RESOURCE
ResType_IO	IO_RESOURCE
ResType_IRQ	IRQ_RESOURCE
ResType_Mem	MEM_RESOURCE
ResType_MfCardConfig	MFCARD_RESOURCE
ResType_PcCardConfig	PCCARD_RESOURCE

ResourceLen

Caller-supplied length of the structure pointed to by *ResourceData*.

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

Callers of **CM_Add_Res_Des** must call **CM_Free_Res_Des_Handle** to deallocate the resource descriptor handle, after it is no longer needed.

See Also

CM_Add_Res_Des_Ex, CM_Free_Res_Des_Handle

CM_Add_Res_Des_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Add_Res_Des_Ex(
    OUT PRES_DES prdResDes,
    IN LOG_CONF lcLogConf,
    IN RESOURCEID ResourceID,
    IN PCVOID ResourceData,
    IN ULONG ResourceLen,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Add_Res_Des_Ex** function adds a resource descriptor to a logical configuration, on either the local or a remote system.

Parameters

prdResDes

Pointer to a location to receive a handle to the new resource descriptor.

lcLogConf

Caller-supplied handle to the logical configuration to which the resource descriptor should be added. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Empty_Log_Conf
CM_Add_Empty_Log_Conf_Ex
CM_Get_First_Log_Conf
CM_Get_First_Log_Conf_Ex
CM_Get_Next_Log_Conf
CM_Get_Next_Log_Conf_Ex

ResourceID

Caller-supplied resource type identifier, which identifies the type of structure supplied by *ResourceData*. This must be one of the **ResType_**-prefixed constants defined in *cfgmgr32.h*.

ResourceData

Caller-supplied pointer to one of the resource structures listed in the following table.

***ResourceID* Parameter**	**Resource Structure**
ResType_BusNumber	BUSNUMBER_RESOURCE
ResType_ClassSpecific	CS_RESOURCE

ResourceID Parameter	Resource Structure
ResType_DevicePrivate	DEVPRIVATE_RESOURCE
ResType_DMA	DMA_RESOURCE
ResType_IO	IO_RESOURCE
ResType_IRQ	IRQ_RESOURCE
ResType_Mem	MEM_RESOURCE
ResType_MfCardConfig	MFCARD_RESOURCE
ResType_PcCardConfig	PCCARD_RESOURCE

ResourceLen

Caller-supplied length of the structure pointed to by *ResourceData.*

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**, or NULL.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

Callers of **CM_Add_Res_Des_Ex** must call **CM_Free_Res_Des_Handle** to deallocate the resource descriptor handle, after it is no longer needed.

See Also

CM_Add_Res_Des, **CM_Free_Res_Des_Handle**

CM_Connect_Machine

```
CMAPI CONFIGRET WINAPI
  CM_Connect_Machine(
    IN PCTSTR UNCServerName,
    OUT PHMACHINE phMachine
    );
```

The **CM_Connect_Machine** function creates a connection to a remote system.

Parameters

UNCServerName

Caller-supplied text string representing the UNC name, including the \\ prefix, of the system for which a connection will be made.

phMachine

Address of a location to receive a machine handle.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

Callers of **CM_Connect_Machine** must call **CM_Disconnect_Machine** to deallocate the machine handle, after it is no longer needed.

CM_Disconnect_Machine

```
CMAPI CONFIGRET WINAPI
  CM_Disconnect_Machine(
    IN HMACHINE hMachine
    );
```

The **CM_Disconnect_Machine** function removes a connection to a remote system.

Parameters

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

CM_Enumerate_Classes

```
CMAPI CONFIGRET WINAPI
  CM_Enumerate_Classes(
    IN ULONG ulClassIndex,
    OUT LPGUID ClassGuid,
    IN ULONG ulFlags
    );
```

The **CM_Enumerate_Classes** function, when called repeatedly, enumerates the local system's installed device classes by supplying each class's GUID.

Parameters

ulClassIndex

Caller-supplied index into the system's list of device classes. For more information, see the following **Comments** section.

ClassGuid

Caller-supplied address of a GUID structure (described in the Platform SDK) to receive a device class's GUID.

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

To enumerate the local system's device classes, call **CM_Enumerate_Classes** repeatedly, starting with a *ulClassIndex* value of zero and incrementing the index value with each subsequent call until the function returns CR_NO_SUCH_VALUE. Some index values might represent list entries containing invalid class data, in which case the function returns CR_INVALID_DATA. This return value can be ignored.

The class GUIDs obtained from this function can be used as input to the device installation functions.

See Also

CM_Enumerate_Classes_Ex

CM_Enumerate_Classes_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Enumerate_Classes_Ex(
    IN ULONG ulClassIndex,
    OUT LPGUID ClassGuid,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Enumerate_Classes_Ex** function, when called repeatedly, enumerates a local or remote system's installed device classes, by supplying each class's GUID.

Parameters

ulClassIndex

Caller-supplied index into the system's list of device classes. For more information, see the following **Comments** section.

ClassGuid

Caller-supplied address of a GUID structure (described in the Platform SDK) to receive a device class's GUID.

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

To enumerate the local or remote system's device classes, call **CM_Enumerate_Classes_Ex** repeatedly, starting with a *ulClassIndex* index value of zero and incrementing the index value with each subsequent call until the function returns CR_NO_SUCH_VALUE. Some index values might represent list entries containing invalid class data, in which case the function returns CR_INVALID_DATA. This return value can be ignored.

The class GUIDs obtained from this function can be used as input to the device installation functions.

See Also

CM_Enumerate_Classes

CM_Enumerate_Enumerators

```
CMAPI CONFIGRET WINAPI
  CM_Enumerate_Enumerators(
    IN ULONG ulEnumIndex,
    OUT PTCHAR Buffer,
IN OUT PULONG pulLength,
IN ULONG ulFlags
);
```

The **CM_Enumerate_Enumerators** function enumerates the local system's device enumerators by supplying each enumerator's name.

Parameters

ulEnumIndex

Caller-supplied index into the system's list of device enumerators. For more information, see the following **Comments** section.

Buffer

Address of a buffer to receive an enumerator name. This buffer should be MAX_DEVICE_ID_LEN-sized (or, set *Buffer* to zero and obtain the actual name length in the location referenced by *puLength*).

pulLength

Caller-supplied address of a location to hold the buffer size. The caller supplies the length of the buffer pointed to by *Buffer*. The function replaces this value with the actual size of the enumerator's name string. If the caller-supplied buffer length is too small, the function supplies the required buffer size and returns CR_BUFFER_SMALL.

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

To enumerate the local system's device enumerators, call **CM_Enumerate_Enumerators** repeatedly, starting with a *ulEnumIndex* index value of zero. and incrementing the index value with each subsequent call until the function returns CR_NO_SUCH_VALUE.

After enumerator names have been obtained, the names can be used as input to **CM_Get_Device_ID_List**.

See Also

CM_Enumerate_Enumerators_Ex

CM_Enumerate_Enumerators_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Enumerate_Enumerators_Ex(
    IN ULONG ulEnumIndex,
    OUT PTCHAR Buffer,
    IN OUT PULONG pulLength,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Enumerate_Enumerators_Ex** function enumerates a local or remote system's device enumerators, by supplying each enumerator's name.

Parameters

ulEnumIndex

Caller-supplied index into the system's list of device enumerators. For more information, see the following **Comments** section.

Buffer

Address of a buffer to receive an enumerator name. This buffer should be MAX_DEVICE_ID_LEN-sized (or, set *Buffer* to zero and obtain the actual name length in the location referenced by *puLength*).

pulLength

Caller-supplied address of a location to hold the buffer size. The caller supplies the length of the buffer pointed to by *Buffer*. The function replaces this value with the actual size of the enumerator's name string. If the caller-supplied buffer length is too small, the function supplies the required buffer size and returns CR_BUFFER_SMALL.

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

To enumerate the local or remote system's device enumerators, call **CM_Enumerate_Enumerators_Ex** repeatedly, starting with a *ulEnumIndex* index value of zero, and incrementing the index value with each subsequent call until the function returns CR_NO_SUCH_VALUE.

After enumerator names have been obtained, the names can be used as input to **CM_Get_Device_ID_List**.

See Also

CM_Enumerate_Enumerators

CM_Free_Log_Conf

```
CMAPI CONFIGRET WINAPI
  CM_Free_Log_Conf(
    IN LOG_CONF lcLogConfToBeFreed,
    IN ULONG ulFlags
    );
```

The **CM_Free_Log_Conf** function removes a logical configuration and all associated resource descriptors, for the local system.

Parameters

lcLogConfToBeFreed

Caller-supplied handle to a logical configuration. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Empty_Log_Conf
CM_Add_Empty_Log_Conf_Ex
CM_Get_First_Log_Conf
CM_Get_First_Log_Conf_Ex
CM_Get_Next_Log_Conf
CM_Get_Next_Log_Conf_Ex

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

Calling **CM_Free_Log_Conf** can cause the handles returned by **CM_Get_First_Log_Conf** and **CM_Get_Next_Log_Conf** to become invalid. Thus if you want to obtain logical configurations after calling **CM_Free_Log_Conf**, your code must call **CM_Get_First_Log_Conf** again and start at the first configuration.

Note that calling **CM_Free_Log_Conf** frees the configuration, but not the configuration's handle. To free the handle, call **CM_Free_Log_Conf_Handle**.

See Also

CM_Free_Log_Conf_Ex

CM_Free_Log_Conf_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Free_Log_Conf_Ex(
    IN LOG_CONF lcLogConfToBeFreed,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Free_Log_Conf_Ex** function removes a logical configuration and all associated resource descriptors, for a local or remote system.

Parameters

lcLogConfToBeFreed

Caller-supplied handle to a logical configuration. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Empty_Log_Conf
CM_Add_Empty_Log_Conf_Ex
CM_Get_First_Log_Conf
CM_Get_First_Log_Conf_Ex
CM_Get_Next_Log_Conf
CM_Get_Next_Log_Conf_Ex

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

Calling **CM_Free_Log_Conf_Ex** can cause the handles returned by **CM_Get_First_Log_Conf_Ex** and **CM_Get_Next_Log_Conf_Ex** to become invalid. Thus if you want to obtain logical configurations after calling **CM_Free_Log_Conf_Ex**, your code must call **CM_Get_First_Log_Conf_Ex** again and start at the first configuration.

Note that calling **CM_Free_Log_Conf_Ex** frees the configuration, but not the configuration's handle. To free the handle, call **CM_Free_Log_Conf_Handle_Ex**.

See Also

CM_Free_Log_Conf

CM_Free_Log_Conf_Handle

```
CMAPI CONFIGRET WINAPI
  CM_Free_Log_Conf_Handle(
    IN LOG_CONF lcLogConf
    );
```

The **CM_Free_Log_Conf_Handle** function invalidates a logical configuration handle and frees its associated memory allocation.

Parameters

lcLogConf

Caller-supplied logical configuration handle. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Empty_Log_Conf
CM_Add_Empty_Log_Conf_Ex
CM_Get_First_Log_Conf
CM_Get_First_Log_Conf_Ex
CM_Get_Next_Log_Conf
CM_Get_Next_Log_Conf_Ex

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

Each time your code calls one of the functions listed under the description of *lcLogConf*, it must subsequently call **CM_Free_Log_Conf_Handle**.

CM_Free_Res_Des

```
CMAPI CONFIGRET WINAPI
  CM_Free_Res_Des(
    OUT PRES_DES prdResDes,
    IN RES_DES rdResDes,
    IN ULONG ulFlags
    );
```

The **CM_Free_Res_Des** function removes a resource descriptor from a logical configuration, for the local system.

Parameters

prdResDes

Caller-supplied location to receive a handle to the configuration's previous resource descriptor. This parameter can be NULL. For more information, see the following **Comments** section.

rdResDes

Caller-supplied handle to the resource descriptor to be removed. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Res_Des
CM_Add_Res_Des_Ex
CM_Get_Next_Res_Des
CM_Get_Next_Res_Des_Ex
CM_Modify_Res_Des
CM_Modify_Res_Des_Ex

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

Resource descriptors for each configuration are stored in an array. If you specify an address for *prdResDes*, then **CM_Free_Res_Des** returns a handle to the resource descriptor that was previous, in the array, to the one removed. If the handle specified by *rdResDes* represents the resource descriptor located first in the array, then *prdResDes* receives a handle to the logical configuration.

Note that calling **CM_Free_Res_Des** frees the resource descriptor, but not the descriptor's handle. To free the handle, call **CM_Free_Res_Des_Handle**.

See Also

CM_Free_Res_Des_Ex

CM_Free_Res_Des_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Free_Res_Des_Ex(
    OUT PRES_DES prdResDes,
    IN RES_DES rdResDes,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Free_Res_Des_Ex** function removes a resource descriptor from a logical configuration for a local or remote system.

Parameters

prdResDes

Caller-supplied location to receive a handle to the configuration's previous resource descriptor. This parameter can be NULL. For more information, see the following **Comments** section.

rdResDes

Caller-supplied handle to the resource descriptor to be removed. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Res_Des
CM_Add_Res_Des_Ex
CM_Get_Next_Res_Des
CM_Get_Next_Res_Des_Ex
CM_Modify_Res_Des
CM_Modify_Res_Des_Ex

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Comments

Resource descriptors for each configuration are stored in an array. If you specify an address for *prdResDes*, then **CM_Free_Res_Des** returns a handle to the resource descriptor that was previous, in the array, to the one removed. If the handle specified by *rdResDes* represents the resource descriptor located first in the array, then *prdResDes* receives a handle to the logical configuration.

Note that calling **CM_Free_Res_Des_Ex** frees the resource descriptor, but not the descriptor's handle. To free the handle, call **CM_Free_Res_Des_Handle_Ex**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

See Also

CM_Free_Res_Des

CM_Free_Res_Des_Handle

```
CMAPI CONFIGRET WINAPI
  CM_Free_Res_Des_Handle(
    IN RES_DES rdResDes
    );
```

The **CM_Free_Res_Des_Handle** function invalidates a resource description handle and frees its associated memory allocation.

Parameters

rdResDes

Caller-supplied resource descriptor handle to be freed. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Res_Des
CM_Add_Res_Des_Ex
CM_Get_Next_Res_Des
CM_Get_Next_Res_Des_Ex

CM_Modify_Res_Des
CM_Modify_Res_Des_Ex

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

Each time your code calls one of the functions listed under the description of *RdResDes*, it must subsequently call **CM_Free_Res_Des_Handle**.

CM_Free_Resource_Conflict_Handle

```
CMAPI CONFIGRET WINAPI
  CM_Free_Resource_Conflict_Handle(
    IN CONFLICT_LIST clConflictList
    );
```

The **CM_Free_Resource_Conflict_Handle** function invalidates a handle to a resource conflict list, and frees the handle's associated memory allocation.

Parameters

clConflictList

Caller-supplied handle to be freed. This conflict list handle must have been previously obtained by calling **CM_Query_Resource_Conflict_List**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

An application must call **CM_Free_Resource_Conflict_Handle** after it has finished using the handle that was obtained calling **CM_Query_Resource_Conflict_List**.

CM_Get_Child

```
CMAPI CONFIGRET WINAPI
  CM_Get_Child(
    OUT PDEVINST pdnDevInst,
    IN DEVINST dnDevInst,
    IN ULONG ulFlags
    );
```

The **CM_Get_Child** function is used to obtain a device instance handle to the first child node of a specified device node, in the local system's device tree.

Parameters

pdnDevInst

Caller-supplied address of a location to receive the first child's device instance handle.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

To enumerate all children of a device node in the local system's device tree, first call **CM_Get_Child** to obtain a device instance handle to the first child node, then call **CM_Get_Sibling** to obtain handles for the rest of the children.

See Also

CM_Get_Child_Ex

CM_Get_Child_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Child_Ex(
    OUT PDEVINST pdnDevInst,
    IN DEVINST dnDevInst,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Child_Ex** function is used to obtain a device instance handle to the first child node of a specified device node, in a local or remote system's device tree.

Parameters

pdnDevInst

Caller-supplied address of a location to receive the first child's device instance handle.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

To enumerate all children of a device node in a local or remote system's device tree, first call **CM_Get_Child_Ex** to obtain a handle to the first child node, then call **CM_Get_Sibling_Ex** to obtain handles for the rest of the children.

See Also

CM_Get_Child

CM_Get_Depth

```
CMAPI CONFIGRET WINAPI
  CM_Get_Depth(
    OUT PULONG pulDepth,
    IN DEVINST dnDevInst,
    IN ULONG ulFlags
    );
```

The **CM_Get_Depth** function is used to obtain the depth of a specified device node, within the local system's device tree.

Parameters

pulDepth

Caller-supplied address of a location to receive a depth value, where zero represents the device tree's root node, one represents the root node's children, and so on.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

See Also

CM_Get_Depth_EX

CM_Get_Depth_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Depth_Ex(
    OUT PULONG pulDepth,
    IN DEVINST dnDevInst,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Depth_Ex** function is used to obtain the depth of a specified device node, within a local or remote system's device tree.

Parameters

pulDepth

Caller-supplied address of a location to receive a depth value, where zero represents the device tree's root node, one represents the root node's children, and so on.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

See Also

CM_Get_Depth

CM_Get_Device_ID

```
CMAPI CONFIGRET WINAPI
  CM_Get_Device_ID(
    IN DEVINST dnDevInst,
    OUT PTCHAR Buffer,
    IN ULONG BufferLen,
    IN ULONG ulFlags
    );
```

The **CM_Get_Device_ID** function is used for obtaining the device instance ID associated with a specified device instance, on the local system.

Parameters

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

Buffer

Address of a buffer to receive a device instance ID string. The required buffer size can be obtained by calling **CM_Get_Device_ID_Size**, then incrementing the received value to allow room for the string's terminating NULL.

BufferLen

Caller-supplied length, in characters, of the buffer specified by *Buffer*.

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

The function appends a NULL terminator to the supplied device instance ID string, unless the buffer is too small to hold the string. In this case, the function supplies as much of the identifier string as will fit into the buffer, and then returns CR_BUFFER_SMALL.

See Also

CM_Get_Device_ID_Ex

CM_Get_Device_ID_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Device_ID_Ex(
    IN DEVINST dnDevInst,
    OUT PTCHAR Buffer,
    IN ULONG BufferLen,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Device_ID_Ex** function is used for obtaining the device instance ID associated with a specified device instance, on a local or remote system.

Parameters

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex

CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

Buffer

Address of a buffer to receive a device instance ID string. The required buffer size can be obtained by calling **CM_Get_Device_ID_Size_Ex**, then incrementing the received value to allow room for the string's terminating NULL.

BufferLen

Caller-supplied length, in characters, of the buffer specified by *Buffer*.

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

The function appends a NULL terminator to the supplied device instance ID string, unless the buffer is too small to hold the string. In this case, the function supplies as much of the identifier string as will fit into the buffer, and then returns CR_BUFFER_SMALL.

See Also

CM_Get_Device_ID

CM_Get_Device_ID_List

```
CMAPI CONFIGRET WINAPI
  CM_Get_Device_ID_List(
    IN PCTSTR pszFilter,    OPTIONAL
    OUT PTCHAR Buffer,
    IN ULONG BufferLen,
    IN ULONG ulFlags
    );
```

The **CM_Get_Device_ID_List** function is used to obtain a list of device instance IDs associated with the local system's device instances.

Parameters

pszFilter

Caller-supplied pointer to a character string specifying a subset of the system's device instance identifiers, or NULL. See the following description of *ulFlags*.

Buffer

Address of a buffer to receive a set of NULL-terminated device instance identifier strings. The end of the set is terminated by an extra NULL. The required buffer size should be obtained by calling **CM_Get_Device_ID_List_Size**.

BufferLen

Caller-supplied length, in characters, of the buffer specified by *Buffer*.

ulFlags

One of the optional, caller-supplied bit flags, listed in the following table, which specify search filters. If no flags are specified, the function returns all device instance IDs for all device instances.

Flag	Definition
CM_GETIDLIST_FILTER_ENUMERATOR	If this flag is set, *pszFilter* must specify the name of a device enumerator, optionally followed by a device identifier. The string format is *EnumeratorName*\\<*DeviceID*>, such as **ROOT** or **ROOT\*PNP0500**. If *pszFilter* supplies only an enumerator name, the function returns device instance IDs for the instances of each device associated with the enumerator. Enumerator names can be obtained by calling **CM_Enumerate_Enumerators**. If *pszFilter* supplies both an enumerator and a device ID, the function returns device instance IDs only for the device instances of the specified device, associated with the enumerator.
CM_GETIDLIST_FILTER_SERVICE	If this flag is set, *pszFilter* must specify the name of a Windows 2000 service (typically a driver). The function returns device instance IDs for the device instances controlled by the specified service. Note that if the device tree does not contain a device node for the specified service, this function creates one by default. To inhibit this behavior, also set CM_GETIDLIST_DONOTGENERATE.
CM_GETIDLIST_FILTER_EJECTRELATIONS	If this flag is set, *pszFilter* must specify a device name. The function returns device instance IDs for the ejection relations of the specified device instance.

Continued

Flag	Definition
CM_GETIDLIST_FILTER_REMOVALRELATIONS	If this flag is set, *pszFilter* must specify a device name. The function returns device instance IDs for the removal relations of the specified device instance.
CM_GETIDLIST_FILTER_POWERRELATIONS	*Not used.*
CM_GETIDLIST_FILTER_BUSRELATIONS	*Not used.*
CM_GETIDLIST_DONOTGENERATE	Used only with CM_GETIDLIST_FILTER_SERVICE. If set, and if the device tree does not contain a device node for the specified service, this flag prevents the function from creating a device node for the service.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

CM_Get_Device_ID_List_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Device_ID_List_Ex(
    IN PCTSTR pszFilter,    OPTIONAL
    OUT PTCHAR Buffer,
    IN ULONG BufferLen,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Device_ID_List_Ex** function is used to obtain a list of device instance identifiers associated with device instances on a local or remote system.

Parameters

pszFilter

Caller-supplied pointer to a character string specifying a subset of the system's device instance identifiers, or NULL. See the following description of *ulFlags*.

Buffer

Address of a buffer to receive a set of NULL-terminated device instance identifier strings. The end of the set is terminated by an extra NULL. The required buffer size should be obtained by calling **CM_Get_Device_ID_List_Size_Ex**.

BufferLen

Caller-supplied length, in characters, of the buffer specified by *Buffer*.

ulFlags

One of the optional, caller-supplied bit flags that specify search filters. If no flags are specified, the function supplies all instance identifiers for all device instances. For a list of bit flags, see the *ulFlags* description for **CM_Get_Device_ID_List**.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

CM_Get_Device_ID_List_Size

```
CMAPI CONFIGRET WINAPI
  CM_Get_Device_ID_List_Size(
    OUT PULONG pulLen,
    IN PCTSTR pszFilter,    OPTIONAL
    IN ULONG ulFlags
    );
```

The **CM_Get_Device_ID_List_Size** function supplies the buffer size required to hold a list of device instance identifiers associated with the local system's device instances.

Parameters

pulLen

Receives a value representing the required buffer size, in characters.

pszFilter

Caller-supplied pointer to a character string specifying a subset of the system's device instance identifiers, or NULL. See the following description of *ulFlags*.

ulFlags

One of the optional, caller-supplied bit flags that specify search filters. If no flags are specified, the function supplies the buffer size required to hold all instance identifiers for all device instances. For a list of bit flags, see the *ulFlags* description for **CM_Get_Device_ID_List**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

The **CM_Get_Device_ID_List_Size** function should be called to determine the buffer size required by **CM_Get_Device_ID_List**.

The size value supplied in the location pointed to by *pulLen* is guaranteed to represent a buffer size large enough to hold all device instance identifier strings and terminating NULLs. The supplied value might actually represent a buffer size that is larger than necessary, so don't assume the value represents the true length of the character strings that **CM_Get_Device_ID_List** will provide.

See Also

CM_Get_Device_ID_List_Size_Ex

CM_Get_Device_ID_List_Size_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Device_ID_List_Size_Ex(
    OUT PULONG pulLen,
    IN PCTSTR pszFilter,    OPTIONAL
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Device_ID_List_Size_Ex** function supplies the buffer size required to hold a list of device instance identifiers associated with a local or remote system's device instances.

Parameters

pulLen

Receives a value representing the required buffer size, in characters.

pszFilter

Caller-supplied pointer to a character string specifying a subset of the system's device instance identifiers, or NULL. See the following description of *ulFlags*.

ulFlags

One of the optional, caller-supplied bit flags that specify search filters. If no flags are specified, the function supplies the buffer size required to hold all instance identifiers for all device instances. For a list of bit flags, see the *ulFlags* description for **CM_Get_Device_ID_List_Ex**.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

The **CM_Get_Device_ID_List_Size_Ex** function should be called to determine the buffer size required by **CM_Get_Device_ID_List_Ex**.

The size value supplied in the location pointed to by *pulLen* is guaranteed to represent a buffer size large enough to hold all device instance identifier strings and terminating NULLs. The supplied value might actually represent a buffer size that is larger than necessary, so don't assume the value represents the true length of the character strings that **CM_Get_Device_ID_List_Ex** will provide.

See Also

CM_Get_Device_ID_List_Size

CM_Get_Device_ID_Size

```
CMAPI CONFIGRET WINAPI
  CM_Get_Device_ID_Size(
    OUT PULONG pulLen,
    IN DEVINST dnDevInst,
    IN ULONG ulFlags
    );
```

The **CM_Get_Device_ID_Size** function supplies the buffer size required to hold a device instance identifier associated with a device instance on the local system.

Parameters

pulLen

Receives a value representing the required buffer size, in characters.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

The **CM_Get_Device_ID_Size** function should be called to determine the buffer size required by **CM_Get_Device_ID**.

The size value supplied in the location pointed to by *pulLen* is less than or equal to MAX_DEVICE_ID_LEN, and does not include the identifier string's terminating NULL. If the specified device instance does not exist, the function supplies a size value of zero.

See Also

CM_Get_Device_ID_Size_Ex

CM_Get_Device_ID_Size_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Device_ID_Size_Ex(
    OUT PULONG pulLen,
    IN DEVINST dnDevInst,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Device_ID_Size_Ex** function supplies the buffer size required to hold a device instance identifier associated with a device instance on a local or remote system.

Parameters

pulLen

Receives a value representing the required buffer size, in characters.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

The **CM_Get_Device_ID_Size_Ex** function should be called to determine the buffer size required by **CM_Get_Device_ID_Ex**.

The size value supplied in the location pointed to by *pulLen* is less than or equal to MAX_ DEVICE_ID_LEN, and does not include the identifier string's terminating NULL. If the specified device instance does not exist, the function supplies a size value of zero.

See Also

CM_Get_Device_ID_Size

CM_Get_DevNode_Status

```
CMAPI CONFIGRET WINAPI
  CM_Get_DevNode_Status(
    OUT PULONG pulStatus,
    OUT PULONG pulProblemNumber,
    IN DEVINST dnDevInst,
    IN ULONG ulFlags
    );
```

The **CM_Get_DevNode_Status** function is used to obtain the status of a device instance from its device node, in the local system's device tree.

Parameters

pulStatus

Address of a location to receive status bit flags. The function can set any combination of the **DN_**-prefixed bit flags defined in *cfg.h.*

pulProblemNumber

Address of a location to receive one of the **CM_PROB_**-prefixed problem values defined in *cfg.h.* Used only if DN_HAS_PROBLEM is set in *pulStatus.*

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

See Also

CM_Get_DevNode_Status_Ex

CM_Get_DevNode_Status_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_DevNode_Status_Ex(
    OUT PULONG pulStatus,
    OUT PULONG pulProblemNumber,
    IN DEVINST dnDevInst,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_DevNode_Status_Ex** function is used to obtain the status of a device instance from its device node, on a local or remote system's device tree.

Parameters

pulStatus

Address of a location to receive status bit flags. The function can set any combination of the **DN_**-prefixed bit flags defined in *cfg.h*.

pulProblemNumber

Address of a location to receive one of the **CM_PROB_**-prefixed problem values defined in *cfg.h*. Used only if DN_HAS_PROBLEM is set in *pulStatus*.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

See Also

CM_Get_DevNode_Status

CM_Get_First_Log_Conf

```
CMAPI CONFIGRET WINAPI
  CM_Get_First_Log_Conf(
    OUT PLOG_CONF plcLogConf,    OPTIONAL
    IN DEVINST dnDevInst,
    IN ULONG ulFlags
    );
```

The **CM_Get_First_Log_Conf** function is used to obtain the first logical configuration, of a specified configuration type, associated with a specified device instance on the local system.

Parameters

plcLogConf

Address of a location to receive the handle to a logical configuration, or NULL. (See the following **Comments** section.)

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Caller-supplied flag value indicating the type of logical configuration being requested. One of the flags in the following table must be specified.

Configuration Type Flags	Definitions
BASIC_LOG_CONF	The caller is requesting basic configuration information.
FILTERED_LOG_CONF	The caller is requesting filtered configuration information.
ALLOC_LOG_CONF	The caller is requesting allocated configuration information.
BOOT_LOG_CONF	The caller is requesting boot configuration information.
FORCED_LOG_CONF	The caller is requesting forced configuration information.
OVERRIDE_LOG_CONF	The caller is requesting override configuration information.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

Calling **CM_Add_Empty_Log_Conf** or **CM_Free_Log_Conf** can invalidate the handle obtained from a previous call to **CM_Get_First_Log_Conf**. Thus if you want to obtain logical configurations after calling **CM_Add_Empty_Log_Conf** or **CM_Free_Log_Conf**, your code must call **CM_Get_First_Log_Conf** again and start at the first configuration.

The handle received in *plcLogConf* must be explicitly freed by calling **CM_Free_Log_Conf_Handle**.

If **CM_Get_First_Log_Conf** is called with *plcLogConf* set to NULL, no handle is returned. This allows you to use the return status to determine if a configuration exists without the need to subsequently free the handle.

See Also

CM_Get_First_Log_Conf_Ex

CM_Get_First_Log_Conf_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_First_Log_Conf_Ex(
    OUT PLOG_CONF plcLogConf,    OPTIONAL
    IN DEVINST dnDevInst,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_First_Log_Conf_Ex** function is used to obtain the first logical configuration associated with a specified device instance, on a local or remote system.

Parameters

plcLogConf

Address of a location to receive the handle to a logical configuration, or NULL. (See the following **Comments** section.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Caller-supplied flag value indicating the type of logical configuration being requested. For a list of flags, see the *ulFlags* description for **CM_Get_First_Log_Conf**.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

Calling **CM_Add_Empty_Log_Conf_Ex** or **CM_Free_Log_Conf_Ex** can invalidate the handle obtained from a previous call to **CM_Get_First_Log_Conf_Ex**. Thus if you want to obtain logical configurations after calling **CM_Add_Empty_Log_Conf_Ex** or **CM_Free_Log_Conf_Ex**, your code must call **CM_Get_First_Log_Conf_Ex** again and start at the first configuration.

The handle received in *plcLogConf* must be explicitly freed by calling **CM_Free_Log_Conf_Handle**.

If **CM_Get_First_Log_Conf_Ex** is called with *plcLogConf* set to NULL, no handle is returned. This allows you to use the return status to determine if a configuration exists without the need to subsequently free the handle.

See Also

CM_Get_First_Log_Conf

CM_Get_Log_Conf_Priority

```
CMAPI CONFIGRET WINAPI
  CM_Get_Log_Conf_Priority(
    IN LOG_CONF lcLogConf,
    OUT PPRIORITY pPriority,
    IN ULONG ulFlags
    );
```

The **CM_Get_Log_Conf_Priority** function is used to obtain the configuration priority of a specified logical configuration, on the local system.

Parameters

lcLogConf

Caller-supplied handle to a logical configuration. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Empty_Log_Conf
CM_Add_Empty_Log_Conf_Ex
CM_Get_First_Log_Conf
CM_Get_First_Log_Conf_Ex
CM_Get_Next_Log_Conf
CM_Get_Next_Log_Conf_Ex

pPriority

Caller-supplied address of a location to receive a configuration priority value. For a list of priority values, see the description of *Priority* for **CM_Add_Empty_Log_Conf**.

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

See Also

CM_Get_Log_Conf_Priority_Ex

CM_Get_Log_Conf_Priority_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Log_Conf_Priority_Ex(
    IN LOG_CONF lcLogConf,
    OUT PPRIORITY pPriority,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Log_Conf_Priority_Ex** function is used to obtain the configuration priority of a specified logical configuration, on a local or remote system.

Parameters

lcLogConf

Caller-supplied handle to a logical configuration. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Empty_Log_Conf
CM_Add_Empty_Log_Conf_Ex
CM_Get_First_Log_Conf
CM_Get_First_Log_Conf_Ex
CM_Get_Next_Log_Conf
CM_Get_Next_Log_Conf_Ex

pPriority

Caller-supplied address of a location to receive a configuration priority value. For a list of priority values, see the description of *Priority* for **CM_Add_Empty_Log_Conf_Ex**.

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

See Also

CM_Get_Log_Conf_Priority

CM_Get_Next_Log_Conf

```
CMAPI CONFIGRET WINAPI
  CM_Get_Next_Log_Conf(
    OUT PLOG_CONF plcLogConf,     OPTIONAL
    IN LOG_CONF lcLogConf,
    IN ULONG ulFlags
    );
```

The **CM_Get_Next_Log_Conf** function is used to obtain the next logical configuration associated with a specific device instance, on the local system.

Parameters

plcLogConf

Address of a location to receive the handle to a logical configuration, or NULL. (See the following **Comments** section.

lcLogConf

Caller-supplied handle to a logical configuration. This handle must have been previously obtained by calling one of the following functions:

CM_Get_First_Log_Conf
CM_Get_Next_Log_Conf

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

To enumerate the logical configurations associated with a device instance, call **CM_Get_First_Log_Conf** to obtain the first logical configuration of a specified configuration type, then call **CM_Get_Next_Log_Conf** repeatedly until it returns CR_NO_MORE_LOG_CONF.

Calling **CM_Add_Empty_Log_Conf** or **CM_Free_Log_Conf** can invalidate the handle obtained from a previous call to **CM_Get_Next_Log_Conf**. Thus if you want to obtain logical configurations after calling **CM_Add_Empty_Log_Conf** or **CM_Free_Log_Conf**, your code must call **CM_Get_First_Log_Conf** again and start at the first configuration.

The handle received in *plcLogConf* must be explicitly freed by calling **CM_Free_Log_Conf_Handle**.

If **CM_Get_Next_Log_Conf** is called with *plcLogConf* set to NULL, no handle is returned. This allows you to use the return status to determine if a configuration exists without the need to subsequently free the handle.

See Also

CM_Get_Next_Log_Conf_Ex

CM_Get_Next_Log_Conf_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Next_Log_Conf_Ex(
    OUT PLOG_CONF plcLogConf,    OPTIONAL
    IN LOG_CONF lcLogConf,
     IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Next_Log_Conf_Ex** function is used to obtain the next logical configuration associated with a specific device instance, on a local or remote system.

Parameters

plcLogConf

Address of a location to receive the handle to a logical configuration, or NULL. (See the following **Comments** section.

lcLogConf

Caller-supplied handle to a logical configuration. This handle must have been previously obtained by calling one of the following functions:

CM_Get_First_Log_Conf_Ex
CM_Get_Next_Log_Conf_Ex

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

To enumerate the logical configurations associated with a device instance, call **CM_Get_First_Log_Conf_Ex** to obtain the first logical configuration, then call **CM_Get_Next_Log_Conf_Ex** repeatedly until it returns CR_NO_MORE_LOG_CONF.

Calling **CM_Add_Empty_Log_Conf_Ex** or **CM_Free_Log_Conf_Ex** can invalidate the handle obtained from a previous call to **CM_Get_Next_Log_Conf_Ex**. Thus if you want to obtain logical configurations after calling **CM_Add_Empty_Log_Conf_Ex** or **CM_Free_Log_Conf_Ex**, your code must call **CM_Get_First_Log_Conf_Ex** again and start at the first configuration.

The handle received in *plcLogConf* must be explicitly freed by calling **CM_Free_Log_Conf_Handle**.

If **CM_Get_Next_Log_Conf_Ex** is called with *plcLogConf* set to NULL, no handle is returned. This allows you to use the return status to determine if a configuration exists without the need to subsequently free the handle.

See Also

CM_Get_Next_Log_Conf

CM_Get_Next_Res_Des

```
CMAPI CONFIGRET WINAPI
  CM_Get_Next_Res_Des(
    OUT PRES_DES prdResDes,
    IN RES_DES rdResDes,
    IN RESOURCEID ForResource,
    OUT PRESOURCEID pResourceID,
    IN ULONG ulFlags
    );
```

The **CM_Get_Next_Res_Des** function is used to obtain a handle to the next resource descriptor, of a specified resource type, for a logical configuration on the local system.

Parameters

prdResDes

Pointer to a location to receive a resource descriptor handle.

rdResDes

Caller-supplied handle to either a resource descriptor or a logical configuration. For more information, see the following **Comments** section.

ForResource

Caller-supplied resource type identifier, indicating the type of resource descriptor being requested. This must be one of the **ResType_**-prefixed constants defined in *cfgmgr32.h.*

pResourceID

Pointer to a location to receive a resource type identifier, if *ForResource* specifies **Res-Type_All**. For any other *ForResource* value, callers should set this to NULL.

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

To enumerate a logical configuration's resource descriptors, begin by calling **CM_Get_ Next_Res_Des** with the logical configuration's handle as the argument for *rdResDes*. This obtains a handle to the first resource descriptor of the type specified by *ForResource*. Then for each subsequent call to **CM_Get_Next_Res_Des**, specify the most recently obtained descriptor handle as the argument for *rdResDes*. Repeat until the function returns CR_NO_MORE_RES_DES.

To retrieve the information stored in a resource descriptor, call **CM_Get_Res_Des_Data**.

To modify the information stored in a resource descriptor, call **CM_Modify_Res_Des**.

Callers of **CM_Get_Next_Res_Des** must call **CM_Free_Res_Des_Handle** to deallocate the resource descriptor handle, after it is no longer needed.

See Also

CM_Get_Next_Res_Des_Ex

CM_Get_Next_Res_Des_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Next_Res_Des_Ex(
    OUT PRES_DES prdResDes,
    IN RES_DES rdResDes,
    IN RESOURCEID ForResource,
    OUT PRESOURCEID pResourceID,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Next_Res_Des_Ex** function is used to obtain a handle to the next resource descriptor, of a specified resource type, for a logical configuration on a local or remote system.

Parameters

prdResDes

Pointer to a location to receive a resource descriptor handle.

rdResDes

Caller-supplied handle to either a resource descriptor or a logical configuration. For more information, see the following **Comments** section.

ForResource

Caller-supplied resource type identifier, indicating the type of resource descriptor being requested. This must be one of the **ResType_**-prefixed constants defined in *cfgmgr32.h*.

pResourceID

Pointer to a location to receive a resource type identifier, if *ForResource* specifies **ResType_All**. For any other *ForResource* value, callers should set this to NULL.

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

To enumerate a logical configuration's resource descriptors, begin by calling **CM_Get_Next_Res_Des_Ex** with the logical configuration's handle as the argument for *rdResDes*. This obtains a handle to the first resource descriptor of the type specified by *ForResource*. Then for each subsequent call to **CM_Get_Next_Res_Des_Ex**, specify the most recently obtained descriptor handle as the argument for *rdResDes*. Repeat until the function returns CR_NO_MORE_RES_DES.

To retrieve the information stored in a resource descriptor, call **CM_Get_Res_Des_Data_Ex**.

To modify the information stored in a resource descriptor, call **CM_Modify_Res_Des_Ex**.

Callers of **CM_Get_Next_Res_Des_Ex** must call **CM_Free_Res_Des_Handle** to deallocate the resource descriptor handle, after it is no longer needed.

See Also

CM_Get_Next_Res_Des

CM_Get_Parent

```
CMAPI CONFIGRET WINAPI
  CM_Get_Parent(
    OUT PDEVINST pdnDevInst,
    IN DEVINST dnDevInst,
    IN ULONG ulFlags
    );
```

The **CM_Get_Parent** function is used to obtain a device instance handle to the parent node of a specified device node, in the local system's device tree.

Parameters

pdnDevInst

Caller-supplied address of a location to receive the first child's device instance handle.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

See Also

CM_Get_Parent_Ex

CM_Get_Parent_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Parent_Ex(
    OUT PDEVINST pdnDevInst,
    IN DEVINST dnDevInst,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Parent_Ex** function is used to obtain a device instance handle to the parent node of a specified device node, in a local or remote system's device tree.

Parameters

pdnDevInst

Caller-supplied address of a location to receive the first child's device instance handle.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

See Also

CM_Get_Parent_Ex

CM_Get_Res_Des_Data

```
CMAPI CONFIGRET WINAPI
  CM_Get_Res_Des_Data(
    IN RES_DES rdResDes,
    OUT PVOID Buffer,
    IN ULONG BufferLen,
    IN ULONG ulFlags
    );
```

The **CM_Get_Res_Des_Data** function is used to retrieve the information stored in a resource descriptor on the local system.

Parameters

rdResDes

Caller-supplied handle to a resource descriptor, obtained by a previous call to **CM_Get_Next_Res_Des**.

Buffer

Address of a buffer to receive the contents of a resource descriptor. The required buffer size should be obtained by calling **CM_Get_Res_Des_Data_Size**.

BufferLen

Caller-supplied length of the buffer specified by *Buffer*.

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

Information returned in the buffer supplied by *Buffer* will be formatted as one of the resource type structures listed in the description of **CM_Add_Res_Des**, based on the resource type that was specified when **CM_Get_Next_Res_Des** was called to obtain the resource descriptor handle.

See Also

CM_Get_Res_Des_Data_Ex

CM_Get_Res_Des_Data_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Res_Des_Data_Ex(
    IN RES_DES rdResDes,
    OUT PVOID Buffer,
    IN ULONG BufferLen,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Res_Des_Data_Ex** function is used to retrieve the information stored in a resource descriptor on a local or remote system.

Parameters

rdResDes

Caller-supplied handle to a resource descriptor, obtained by a previous call to **CM_Get_Next_Res_Des_Ex**.

Buffer

Address of a buffer to receive the contents of a resource descriptor. The required buffer size should be obtained by calling **CM_Get_Res_Des_Data_Size_Ex**.

BufferLen

Caller-supplied length of the buffer specified by *Buffer*.

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

Information returned in the buffer supplied by *Buffer* will be formatted as one of the resource type structures listed in the description of **CM_Add_Res_Des_Ex**, based on the resource type that was specified when **CM_Get_Next_Res_Des_Ex** was called to obtain the resource descriptor handle.

See Also

CM_Get_Res_Des_Data

CM_Get_Res_Des_Data_Size

```
CMAPI CONFIGRET WINAPI
  CM_Get_Res_Des_Data_Size(
    OUT PULONG pulSize,
    IN RES_DES rdResDes,
    IN ULONG ulFlags
    );
```

The **CM_Get_Res_Des_Data_Size** function supplies the buffer size required to hold the information contained in a specified resource descriptor on the local system.

Parameters

pulSize

Caller-supplied address of a location to receive the required buffer size.

rdResDes

Caller-supplied handle to a resource descriptor, obtained by a previous call to **CM_Get_Next_Res_Des**.

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

The returned size value represents the size of the appropriate resource structure (see ***CM_Add_Res_Des***). If the resource descriptor resides in a resource requirements list, the returned size includes both the size of the resource structure and the space allocated for associated range arrays.

See Also

CM_Get_Res_Des_Data_Size_Ex

CM_Get_Res_Des_Data_Size_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Get_Res_Des_Data_Size_Ex(
    OUT PULONG pulSize,
    IN RES_DES rdResDes,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Get_Res_Des_Data_Size_Ex** function supplies the buffer size required to hold the information contained in a specified resource descriptor on a local or remote system.

Parameters

pulSize

Caller-supplied address of a location to receive the required buffer size.

rdResDes

Caller-supplied handle to a resource descriptor, obtained by a previous call to **CM_Get_Next_Res_Des_Ex**.

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

The returned size value represents the size of the appropriate resource structure (see ***CM_Add_Res_Des_Ex***). If the resource descriptor resides in a resource requirements list, the returned size includes both the size of the resource structure and the space allocated for associated range arrays.

See Also

CM_Get_Res_Des_Data_Size

CM_Get_Resource_Conflict_Count

```
CMAPI CONFIGRET WINAPI
  CM_Get_Resource_Conflict_Count(
    IN CONFLICT_LIST clConflictList,
    OUT PULONG pulCount
    );
```

The **CM_Get_Resource_Conflict_Count** function supplies the number of conflicts contained in a specified resource conflict list.

Parameters

clConflictList

Caller-supplied handle to a conflict list, obtained by a previous call to **CM_Query_ Resource_Conflict_List**.

pulCount

Caller-supplied address of a location to receive the conflict count.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

The count value obtained by calling **CM_Get_Resource_Conflict_Count** can be used to determine the number of times to call **CM_Get_Resource_Conflict_Details**, which supplies information about each conflict.

If there are no entries in the conflict list, the location supplied by *pulCount* will receive zero.

CM_Get_Resource_Conflict_Details

```
CMAPI CONFIGRET WINAPI
  CM_Get_Resource_Conflict_Details(
    IN CONFLICT_LIST clConflictList,
    IN ULONG ulIndex,
    IN OUT PCONFLICT_DETAILS pConflictDetails
    );
```

The **CM_Get_Resource_Conflict_Details** function supplies details about one of the resource conflicts in a conflict list.

Parameters

clConflictList

Caller-supplied handle to a conflict list, obtained by a previous call to **CM_Query_ Resource_Conflict_List**.

ulIndex

Caller-supplied value used as an index into the conflict list. This value can be from zero to one less than the number returned by **CM_Get_Resource_Conflict_Count**.

pConflictDetails

Caller-supplied address of a CONFLICT_DETAILS structure to receive conflict details. The caller must supply values for the structure's *CD_ulSize* and *CD_ulMask* structures.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

To determine conflicting resource requirements between a specified device and other devices on a system, use the following steps.

1. Call CM_Query_Resource_Conflict_List to obtain a handle to a list of resource conflicts.
2. Call CM_Get_Resource_Conflict_Count to determine the number of conflicts contained in the resource conflict list.
3. Call CM_Get_Resource_Conflict_Details for each entry in the conflict list.

The following conflicts are typically not reported:

- If there are multiple conflicts for a resource, and the owners of only some of the conflicts can be determined, the conflicts without identifiable owners are not reported.
- Conflicts that appear to be with the specified device (that is, the device conflicts with itself) are not reported.
- If multiple non-Plug and Play devices use the same driver, resource conflicts among these devices might not be reported.

Sometimes, resources assigned to the HAL might be reported as either conflicting with the HAL or not available.

CM_Get_Sibling

```
CMAPI CONFIGRET WINAPI
  CM_Get_Sibling(
    OUT PDEVINST pdnDevInst,
    IN DEVINST DevInst,
    IN ULONG ulFlags
    );
```

The **CM_Get_Sibling** function is used to obtain a device instance handle to the next sibling node of a specified device node, in the local system's device tree.

Parameters

pdnDevInst

Caller-supplied address of a location to receive the sibling's device instance handle.

DevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Get_Child
CM_Get_Child_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

To enumerate all children of a device node in the local system's device tree, first call **CM_Get_Child** to obtain a handle to the first child node, then call **CM_Get_Sibling** to obtain handles for the rest of the children.

See Also

CM_Get_Sibling_Ex

CM_Get_Sibling_Ex

```
MAPI CONFIGRET WINAPI
 CM_Get_Sibling_Ex(
   OUT PDEVINST pdnDevInst,
   IN DEVINST DevInst,
   IN ULONG ulFlags,
   IN HMACHINE hMachine
   );
```

The **CM_Get_Sibling_Ex** function is used to obtain a device instance handle to the next sibling node of a specified device node, in a local or remote system's device tree.

Parameters

pdnDevInst

Caller-supplied address of a location to receive the sibling's device instance handle.

DevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Get_Child
CM_Get_Child_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

To enumerate all children of a device node in the local system's device tree, first call **CM_ Get_Child_Ex** to obtain a handle to the first child node, then call **CM_Get_Sibling_Ex** to obtain handles for the rest of the children.

See Also

CM_Get_Sibling

CM_Get_Version

```
CMAPI WORD WINAPI
  CM_Get_Version(
    VOID
    );
```

The **CM_Get_Version** function returns the version number of *cfgmgr32.dll* residing on the local system.

Return Value

The major revision number is returned in the high byte and the minor revision number is returned in the low byte. For example, version 4.0 is returned as 0x0400.

See Also

CM_Get_Version_Ex

CM_Get_Version_Ex

```
CMAPI WORD WINAPI
  CM_Get_Version_Ex(
    IN HMACHINE hMachine
    );
```

The **CM_Get_Version** function returns the version number of *cfgmgr32.dll* residing on a local or remote system.

Parameters

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

The major revision number is returned in the high byte and the minor revision number is returned in the low byte. For example, version 4.0 is returned as 0x0400.

See Also

CM_Get_Version

CM_Locate_DevNode

```
CMAPI CONFIGRET WINAPI
  CM_Locate_DevNode(
    OUT PDEVINST pdnDevInst,
    IN DEVINSTID pDeviceID,     OPTIONAL
    IN ULONG ulFlags
    );
```

The **CM_Locate_DevNode** function supplies a device instance handle to the device node that is associated with a specified device instance identifier, on the local system.

Parameters

pdnDevInst

Caller-supplied address of a location to receive a device instance handle.

pDeviceID

Caller-supplied pointer to a NULL-terminated string representing a device instance identifier. If this value is NULL, or if it points to a zero-length string, the function supplies a device instance handle to the device node at the top of the device tree.

ulFlags

One of the caller-supplied bit flags listed in the following table.

Flag	Definition
CM_LOCATE_DEVNODE_NORMAL	(*Default.*) The function only searches for device nodes representing devices that are currently plugged into the system.
CM_LOCATE_DEVNODE_PHANTOM	The function searches both for device nodes representing devices that are currently plugged into the system, and for device nodes representing devices that are not currently plugged in (but were previously).
CM_LOCATE_DEVNODE_CANCELREMOVE	*Not used.*
CM_LOCATE_DEVNODE_NOVALIDATION	*Not used.*

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

See Also

CM_Locate_DevNode_Ex

CM_Locate_DevNode_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Locate_DevNode_Ex(
    OUT PDEVINST pdnDevInst,
    IN DEVINSTID pDeviceID,    OPTIONAL
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Locate_DevNode_Ex** function supplies a device instance handle to the device node that is associated with a specified device instance identifier, on a local or remote system.

Parameters

pdnDevInst

Caller-supplied address of a location to receive a device instance handle.

pDeviceID

Caller-supplied pointer to a NULL-terminated string representing a device instance identifier. If this value is NULL, or if it points to a zero-length string, the function supplies a device instance handle to the device node at the top of the device tree.

ulFlags

One of the caller-supplied bit flags listed in the following table.

Flag	Definition
CM_LOCATE_DEVNODE_NORMAL	(*Default.*) The function only searches for device nodes representing devices that are currently plugged into the system.
CM_LOCATE_DEVNODE_PHANTOM	The function searches both for device nodes representing devices that are currently plugged into the system, and for device nodes representing devices that are not currently plugged in (but were previously).
CM_LOCATE_DEVNODE_CANCELREMOVE	*Not used.*
CM_LOCATE_DEVNODE_NOVALIDATION	*Not used.*

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

See Also

CM_Locate_DevNode

CM_Modify_Res_Des

```
CMAPI CONFIGRET WINAPI
  CM_Modify_Res_Des(
    OUT PRES_DES prdResDes,
    IN RES_DES rdResDes,
    IN RESOURCEID ResourceID,
    IN PCVOID ResourceData,
    IN ULONG ResourceLen,
    IN ULONG ulFlags
    );
```

The **CM_Modify_Res_Des** function modifies a specified resource descriptor on the local system.

Parameters

prdResDes

Pointer to a location to receive a handle to the modified resource descriptor.

rdResDes

Caller-supplied handle to the resource descriptor to be modified. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Res_Des
CM_Add_Res_Des_Ex
CM_Get_Next_Res_Des
CM_Get_Next_Res_Des_Ex
CM_Modify_Res_Des
CM_Modify_Res_Des_Ex

ResourceID

Caller-supplied resource type identifier. This must be one of the **ResType_**-prefixed constants defined in *cfgmgr32.h.*

ResourceData

Caller-supplied pointer to a resource descriptor, which can be one of the structures listed under the **CM_Add_Res_Des** function's description of *ResourceData.*

ResourceLen

Caller-supplied length of the structure pointed to by *ResourceData.*

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

The caller-supplied resource descriptor data replaces the existing data. The values specified for *ResourceID* and *ResourceLen* do not have to match the existing resource descriptor.

If the value specified for *ResourceID* is **ResType_ClassSpecific**, then the specified resource descriptor must be the last one associated with the logical configuration.

Callers of **CM_Modify_Res_Des** must call **CM_Free_Res_Des_Handle** to deallocate the resource descriptor handle, after it is no longer needed.

See Also

CM_Modify_Res_Des_Ex

CM_Modify_Res_Des_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Modify_Res_Des_Ex(
    OUT PRES_DES prdResDes,
    IN RES_DES rdResDes,
    IN RESOURCEID ResourceID,
    IN PCVOID ResourceData,
    IN ULONG ResourceLen,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Modify_Res_Des_Ex** function modifies a specified resource descriptor on a local or remote system.

Parameters

prdResDes

Pointer to a location to receive a handle to the modified resource descriptor.

rdResDes

Caller-supplied handle to the resource descriptor to be modified. This handle must have been previously obtained by calling one of the following functions:

CM_Add_Res_Des
CM_Add_Res_Des_Ex
CM_Get_Next_Res_Des
CM_Get_Next_Res_Des_Ex
CM_Modify_Res_Des
CM_Modify_Res_Des_Ex

ResourceID

Caller-supplied resource type identifier. This must be one of the **ResType_**-prefixed constants defined in *cfgmgr32.h.*

ResourceData

Caller-supplied pointer to a resource descriptor, which can be one of the structures listed under the **CM_Add_Res_Des_Ex** function's description of *ResourceData.*

ResourceLen

Caller-supplied length of the structure pointed to by *ResourceData.*

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

The caller-supplied resource descriptor data replaces the existing data. The values specified for *ResourceID* and *ResourceLen* do not have to match the existing resource descriptor.

If the value specified for *ResourceID* is **ResType_ClassSpecific**, then the specified resource descriptor must be the last one associated with the logical configuration.

Callers of **CM_Modify_Res_Des_Ex** must call **CM_Free_Res_Des_Handle** to deallocate the resource descriptor handle, after it is no longer needed.

See Also

CM_Modify_Res_Des

CM_Query_Resource_Conflict_List

```
CMAPI CONFIGRET WINAPI
  CM_Query_Resource_Conflict_List(
    OUT PCONFLICT_LIST pclConflictList,
    IN DEVINST dnDevInst,
    IN RESOURCEID ResourceID,
    IN PCVOID ResourceData,
    IN ULONG ResourceLen,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Query_Resource_Conflict_List** function identifies device instances having resource requirements that conflict with a specified device instance's resource description.

Parameters

pclConflictList

Caller-supplied address of a location to receive a handle to a conflict list.

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ResourceID

Caller-supplied resource type identifier. This must be one of the **ResType_**-prefixed constants defined in *cfgmgr32.h.*

ResourceData

Caller-supplied pointer to a resource descriptor, which can be one of the structures listed under the **CM_Add_Res_Des** function's description of *ResourceData*.

ResourceLen

Caller-supplied length of the structure pointed to by *ResourceData*.

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

When calling **CM_Query_Resource_Conflict_List**, specify a device instance handle and resource descriptor. (Resource descriptors for existing device nodes can be obtained by calling **CM_Get_Res_Des_Data**.) These parameters indicate the specific resources you'd like a specific device to use. The resulting conflict list identifies devices that use the same resources, along with resources reserved by the system.

After calling **CM_Query_Resource_Conflict_List**, an application can call **CM_Get_ Resource_Conflict_Count** to determine the number of conflicts contained in the resource conflict list. (The number of conflicts can be zero.) Then the application can call **CM_Get_ Resource_Conflict_Details** for each entry in the conflict list.

After an application has finished using the handle received for *pclConflictList*, it must call **CM_Free_Resource_Conflict_handle**.

CM_Reenumerate_DevNode

```
CMAPI CONFIGRET WINAPI
  CM_Reenumerate_DevNode(
    IN DEVINST dnDevInst,
    IN ULONG ulFlags
    );
```

The **CM_Reenumerate_DevNode** function enumerates the devices identified by a specified device node and all of its children.

Parameters

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h.*

Comments

If the specified device node represents a hardware or software bus device, the PnP Manager queries the device's drivers for a list of children, then attempts to configure and start any child devices that were not previously configured. The PnP Manager also initiates surprise-removal of devices that are no longer present (see IRP_MN_SURPRISE_REMOVAL).

See Also

CM_Reenumerate_DevNode_Ex

CM_Reenumerate_DevNode_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Reenumerate_DevNode_Ex(
    IN DEVINST dnDevInst,
    IN ULONG ulFlags,
    IN HMACHINE hMachine
    );
```

The **CM_Reenumerate_DevNode_Ex** function enumerates the devices identified by a specified device node and all of its children.

Parameters

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

ulFlags

Not used, must be zero.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

If the specified device node represents a hardware or software bus device, the PnP Manager queries the device's drivers for a list of children, then attempts to configure and start any child devices that were not previously configured. The PnP Manager also initiates surprise-removal of devices that are no longer present (see IRP_MN_SURPRISE_REMOVAL).

See Also

CM_Reenumerate_DevNode

CM_Request_Device_Eject

```
CMAPI CONFIGRET WINAPI
  CM_Request_Device_Eject(
    IN DEVINST dnDevInst,
    OUT PPNP_VETO_TYPE pVetoType,
    OUT LPTSTR pszVetoName,
    IN ULONG ulNameLength,
    IN ULONG ulFlags
    );
```

The **CM_Request_Device_Eject** function prepares a local device instance for safe removal, if the device is removable. If the device can be physically ejected, it will be.

Parameters

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

pVetoType

(*Optional.*) If the removal request fails, this parameter receives a PNP_VETO_TYPE-typed value indicating the reason for the failure.

pszVetoName

(*Optional.*) Caller-supplied pointer to a string buffer that receives a text string. The type of information this string provides is dependent on the value received by *pVetoType*, as indicated in the following table.

pVetoType	pszVetoName
PNP_VetoTypeUnknown	Not used.
PNP_VetoLegacyDevice	A device instance path.
PNP_VetoPendingClose	A device instance path.
PNP_VetoWindowsApp	An application module.
PNP_VetoWindowsService	A Windows service
PNP_VetoOutstandingOpen	A device instance path.
PNP_VetoDevice	A device instance path.
PNP_VetoDriver	A driver name.

Continued

pVetoType	pszVetoName
PNP_VetoIllegalDeviceRequest	A device instance path.
PNP_VetoInsufficientPower	Not used.
PNP_VetoNonDisableable	A device instance path.
PNP_VetoLegacyDriver	A Windows service

ulNameLength

(*Optional.*) Caller-supplied value representing the length of the string buffer supplied by *pszVetoName*. This should be set to MAX_PATH.

ulFlags

Not used.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

See Also

CM_Request_Device_Eject_Ex

CM_Request_Device_Eject_Ex

```
CMAPI CONFIGRET WINAPI
  CM_Request_Device_Eject_Ex(
    IN DEVINST dnDevInst,
    OUT PPNP_VETO_TYPE pVetoType,
    OUT LPTSTR pszVetoName,
    IN ULONG ulNameLength,
    IN ULONG ulFlags
    IN HMACHINE hMachine
    );
```

The **CM_Request_Device_Eject** function prepares a local or remote device instance for safe removal, if the device is removable. If the device can be physically ejected, it will be.

Parameters

dnDevInst

Caller-supplied device instance handle, obtained from the SP_DEVINFO_DATA structure that is used with the device installation functions.

Device instance handles can also be obtained by calling the following functions:

CM_Locate_DevNode
CM_Locate_DevNode_Ex
CM_Get_Child
CM_Get_Child_Ex
CM_Get_Parent
CM_Get_Parent_Ex
CM_Get_Sibling
CM_Get_Sibling_Ex

pVetoType

(*Optional.*) If the removal request fails, this parameter receives a PNP_VETO_TYPE-typed value indicating the reason for the failure.

pszVetoName

(*Optional.*) Caller-supplied pointer to a string buffer that receives a text string. The type of information this string provides is dependent on the value received by *pVetoType*, as indicated in the following table.

pVetoType	pszVetoName
PNP_VetoTypeUnknown	Not used.
PNP_VetoLegacyDevice	A device instance path.
PNP_VetoPendingClose	A device instance path.
PNP_VetoWindowsApp	An application module.
PNP_VetoWindowsService	A Windows service
PNP_VetoOutstandingOpen	A device instance path.
PNP_VetoDevice	A device instance path.
PNP_VetoDriver	A driver name.
PNP_VetoIllegalDeviceRequest	A device instance path.
PNP_VetoInsufficientPower	Not used.
PNP_VetoNonDisableable	A device instance path.
PNP_VetoLegacyDriver	A Windows service

ulNameLength

(*Optional.*) Caller-supplied value representing the length of the string buffer supplied by *pszVetoName*. This should be set to MAX_PATH.

ulFlags

Not used.

hMachine

Caller-supplied machine handle, obtained from a previous call to **CM_Connect_Machine**.

Return Value

If the operation succeeds, the function returns CR_SUCCESS. Otherwise, it returns one of the CR_-prefixed error codes defined in *cfgmgr32.h*.

Comments

For remote systems, this function only works for "dock" device instances. That is, the function can only be used remotely to undock a system. In that case, the caller must have *SeUndockPrivilege*.

See Also

CM_Request_Device_Eject

CHAPTER 7

PnP Configuration Manager Structures and Types

This chapter describes the structures and types used with PnP Configuration Manager functions.

Resource Descriptor Structures

This section describes the structures used to specify resource descriptors.

BUSNUMBER_DES

```
typedef struct BusNumber_Des_s {
  DWORD BUSD_Count;
  DWORD BUSD_Type;
  DWORD BUSD_Flags;
  ULONG BUSD_Alloc_Base;
  ULONG BUSD_Alloc_End;
} BUSNUMBER_DES, *PBUSNUMBER_DES;
```

The BUSNUMBER_DES structure is used for specifying either a resource list or a resource requirements list that describes bus number usage for a device instance.

Members

BUSD_Count

For a resource list:

Zero.

For a resource requirements list:

The number of elements in the BUSNUMBER_RANGE array that is included in the BUSNUMBER_RESOURCE structure.

BUSD_Type

Must be set to the constant value **BusNumberType_Range**.

BUSD_Flags

Not used.

BUSD_Alloc_Base

For a resource list:

The lowest-numbered of a range of contiguous bus numbers allocated to the device.

For a resource requirements list:

Zero.

BUSD_Alloc_End

For a resource list:

The highest-numbered of a range of contiguous bus numbers allocated to the device.

For a resource requirements list:

Zero.

Comments

The BUSNUMBER_DES structure is included as a member of the BUSNUMBER_RESOURCE structure.

BUSNUMBER_RANGE

```
typedef struct BusNumber_Range_s {
  ULONG BUSR_Min;
  ULONG BUSR_Max;
  ULONG BUSR_nBusNumbers;
  ULONG BUSR_Flags;
} BUSNUMBER_RANGE, *PBUSNUMBER_RANGE;
```

The BUSNUMBER_RANGE structure specifies a resource requirements list that describes bus number usage for a device instance.

Members

BUSR_Min

The lowest-numbered of a range of contiguous bus numbers that can be allocated to the device.

BUSR_Max

The highest-numbered of a range of contiguous bus numbers that can be allocated to the device.

BUSR_nBusNumbers

The number of contiguous bus numbers required by the device.

BUSR_Flags

Not used.

Comments

The BUSNUMBER_RANGE structure is included as a member of the BUSNUMBER_RESOURCE structure.

BUSNUMBER_RESOURCE

```
typedef struct BusNumber_Resource_s {
  BUSNUMBER_DES      BusNumber_Header;
  BUSNUMBER_RANGE    BusNumber_Data[ANYSIZE_ARRAY];
} BUSNUMBER_RESOURCE, *PBUSNUMBER_RESOURCE;
```

The BUSNUMBER_RESOURCE structure specifies either a resource list or a resource requirements list that describes bus number usage for a device instance.

Members

BusNumber_Header

A BUSNUMBER_DES structure.

BusNumber_Data

For a resource list:

Zero.

For a resource requirements list:

A BUSNUMBER_RANGE array.

CS_DES

```
typedef struct CS_Des_s {
  DWORD    CSD_SignatureLength;
  DWORD    CSD_LegacyDataOffset;
  DWORD    CSD_LegacyDataSize;
  DWORD    CSD_Flags;
  GUID     CSD_ClassGuid;
  BYTE     CSD_Signature[ANYSIZE_ARRAY];
} CS_DES, *PCS_DES;
```

The CS_DES structure is used for specifying a resource list that describes device class-specific resource usage for a device instance.

Members

CSD_SignatureLength

The number of elements in the byte array specified by **CSD_Signature**.

CSD_LegacyDataOffset

Offset, in bytes, from the beginning of the **CSD_Signature** array to the beginning of a block of data. For example, if the data block follows the signature array, and if the signature array length is 16 bytes, then the value for **CSD_LegacyDataOffset** should be 16.

CSD_LegacyDataSize

Length, in bytes, of the data block whose offset is specified by **CSD_LegacyDataOffset**.

CSD_Flags

Not used.

CSD_ClassGuid

A globally unique identifier (GUID) identifying a device setup class. If both **CSD_SignatureLength** and **CSD_LegacyDataSize** are zero, the GUID is null.

CSD_Signature

A byte array containing a class-specific signature.

Comments

The data block identified by **CSD_LegacyDataSize** and **CSD_LegacyDataOffset** can contain legacy, class-specific data, as stored in the **DeviceSpecificData** member of a CM_PARTIAL_RESOURCE_DESCRIPTOR structure, if the structure's **Type** member is **CmResourceTypeDeviceSpecific**.

The class-specific signature identified by **CSD_SignatureLength** and **CSD_Signature** can contain additional class-specific device identification information.

CS_RESOURCE

```
typedef struct CS_Resource_s {
   CS_DES    CS_Header;
} CS_RESOURCE, *PCS_RESOURCE;
```

The CS_RESOURCE structure is used for specifying a resource list that describes device class-specific resource usage for a device instance.

Members

CS_Header

A CS_DES structure.

DEVPRIVATE_DES

```
typedef struct DevPrivate_Des_s {
  DWORD    PD_Count;
  DWORD    PD_Type;
  DWORD    PD_Data1;
  DWORD    PD_Data2;
  DWORD    PD_Data3;
  DWORD    PD_Flags;
} DEVPRIVATE_DES, *PDEVPRIVATE_DES;
```

The DEVPRIVATE_DES structure is used for specifying either a resource list or a resource requirements list that describes private device-specific resource usage for a device instance. *This structure is for internal use only.*

DEVPRIVATE_RANGE

```
typedef struct DevPrivate_Range_s {
   DWORD    PR_Data1;     // mask for base alignment
   DWORD    PR_Data2;     // number of bytes
   DWORD    PR_Data3;     // minimum address
} DEVPRIVATE_RANGE, *PDEVPRIVATE_RANGE;
```

The DEVPRIVATE_RANGE structure specifies a resource requirements list that describes private device-specific resource usage for a device instance. *This structure is for internal use only.*

DEVPRIVATE_RESOURCE

```
typedef struct DevPrivate_Resource_s {
  DEVPRIVATE_DES      PRV_Header;
  DEVPRIVATE_RANGE    PRV_Data[ANYSIZE_ARRAY];
} DEVPRIVATE_RESOURCE, *PDEVPRIVATE_RESOURCE;
```

The DEVPRIVATE_RESOURCE structure is used for specifying either a resource list or a resource requirements list that describes private device-specific resource usage for a device instance. *This structure is for internal use only.*

DMA_DES

```
typedef struct DMA_Des_s {
  DWORD    DD_Count;
  DWORD    DD_Type;
  DWORD    DD_Flags;
  ULONG    DD_Alloc_Chan;
} DMA_DES, *PDMA_DES;
```

The DMA_DES structure is used for specifying either a resource list or a resource requirements list that describes direct memory access (DMA) channel usage for a device instance.

Members

DD_Count

For a resource list:

Zero.

For a resource requirements list:

The number of elements in the DMA_RANGE array that is included in the DMA_RESOURCE structure.

DD_Type

Must be set to the constant value **DType_Range**.

DD_Flags

One bit flag from *each* of the flag sets described in the following table.

Flag	Definition
Channel Width Flags	
fDD_BYTE	8-bit DMA channel.
fDD_WORD	16-bit DMA channel.
fDD_DWORD	32-bit DMA channel.

	Flag	Definition
	fDD_BYTE_AND_WORD	8-bit and 16-bit DMA channel.
	mDD_Width	Bit mask for the bits within **DD_Flags** that specify the channel width value.
Bus Mastering Flags		
	fDD_NoBusMaster	No bus mastering.
	fDD_BusMaster	Bus mastering.
	mDD_BusMaster	Bit mask for the bits within **DD_Flags** that specify the bus mastering value.
DMA Type Flags		
	fDD_TypeStandard	Standard DMA.
	fDD_TypeA	Type A DMA.
	fDD_TypeB	Type B DMA.
	fDD_TypeF	Type F DMA.
	mDD_Type	Bit mask for the bits within **DD_Flags** that specify the DMA type value.

DD_Alloc_Chan

For a resource list:

The DMA channel allocated to the device.

For a resource requirements list:

Not used.

DMA_RANGE

```
typedef struct DMA_Range_s {
  ULONG    DR_Min;
  ULONG    DR_Max;
  ULONG    DR_Flags;
} DMA_RANGE, *PDMA_RANGE;
```

The DMA_RANGE structure specifies a resource requirements list that describes DMA channel usage for a device instance.

Members

DR_Min

The lowest-numbered DMA channel that can be allocated to the device.

DR_Max

The highest-numbered DMA channel that can be allocated to the device.

DR_Flags

One bit flag from *each* of the flag sets described in the table included with the description of the **DR_Flags** member of the DMA_DES structure.

DMA_RESOURCE

```
typedef struct DMA_Resource_s {
  DMA_DES      DMA_Header;
  DMA_RANGE    DMA_Data[ANYSIZE_ARRAY];
} DMA_RESOURCE, *PDMA_RESOURCE;
```

The DMA_RESOURCE structure is used for specifying either a resource list or a resource requirements list that describes DMA channel usage for a device instance.

Members

DMA_Header

A DMA_DES structure.

DMA_Data

For a resource list:

Zero.

For a resource requirements list:

A DMA_RANGE array.

IO_DES

```
typedef struct IO_Des_s {
  DWORD        IOD_Count;
  DWORD        IOD_Type;
  DWORDLONG    IOD_Alloc_Base;
  DWORDLONG    IOD_Alloc_End;
  DWORD        IOD_DesFlags;
} IO_DES, *PIO_DES;
```

The IO_DES structure is used for specifying either a resource list or a resource requirements list that describes I/O port usage for a device instance.

Members

IOD_Count

For a resource list:

Zero.

For a resource requirements list:

The number of elements in the IO_RANGE array that is included in the IO_RESOURCE structure.

IOD_Type

Must be set to the constant value **IOType_Range**.

IOD_Alloc_Base

For a resource list:

The lowest-numbered of a range of contiguous I/O port addresses allocated to the device.

For a resource requirements list:

Zero.

IOD_Alloc_End

For a resource list:

The highest-numbered of a range of contiguous I/O port addresses allocated to the device.

For a resource requirements list:

Zero.

IOD_DesFlags

One bit flag from *each* of the flag sets described in the following table.

	Flag	Definition
Port Type Flags		
	fIOD_IO	The device is accessed in I/O address space.
	fIOD_Memory	The device is accessed in memory address space.
	fIOD_PortType	Bit mask for the bits within **IOD_DesFlags** that specify the port type value.

Continued

	Flag	Definition
Decode Flags		
	fIOD_10_BIT_DECODE	The device decodes 10 bits of the port address.
	fIOD_12_BIT_DECODE	The device decodes 12 bits of the port address.
	fIOD_16_BIT_DECODE	The device decodes 16 bits of the port address.
	fIOD_POSITIVE_DECODE	The device uses "positive decode" instead of "subtractive decode."
	fIOD_DECODE	Bit mask for the bits within **IOD_DesFlags** that specify the decode value.

IO_RANGE

```
typedef struct IO_Range_s {
  DWORDLONG    IOR_Align;
  DWORD        IOR_nPorts;
  DWORDLONG    IOR_Min;
  DWORDLONG    IOR_Max;
  DWORD        IOR_RangeFlags;
  DWORDLONG    IOR_Alias;
} IO_RANGE, *PIO_RANGE;
```

The IO_RANGE structure specifies a resource requirements list that describes I/O port usage for a device instance.

Members

IOR_Align

Mask used to specify the port address boundary on which the first allocated I/O port address must be aligned.

IOR_nPorts

The number of I/O port addresses required by the device.

IOR_Min

The lowest-numbered of a range of contiguous I/O port addresses that can be allocated to the device.

IOR_Max

The highest-numbered of a range of contiguous I/O port addresses that can be allocated to the device.

IOR_RangeFlags

One bit flag from *each* of the flag sets described in the table included with the description of the **IOD_DesFlags** member of the IO_DES structure. For more information, see the following **Comments** section.

IOR_Alias

One of the bit flags described in the following table.

Flag	Definition
IO_ALIAS_10_BIT_DECODE	The device decodes 10 bits of the port address.
IO_ALIAS_12_BIT_DECODE	The device decodes 12 bits of the port address.
IO_ALIAS_16_BIT_DECODE	The device decodes 16 bits of the port address.
IO_ALIAS_POSITIVE_DECODE	The device uses "positive decode" instead of "subtractive decode."

For more information, see the following **Comments** section.

Comments

The flags specified for **IOR_Alias** have the same interpretation as the address decoding flags specified for **IOD_DesFlags**. (However, the two sets of flags are not equivalent in assigned values and cannot be used interchangeably.) A resource requirements list can be specified using either set of flags, but using decode flags in **IOD_DesFlags** is recommended. If address decoding flags are specified using *both* **IOD_DesFlags** and **IOR_Alias**, contents of the latter overrides the former.

IO_RESOURCE

```
typedef struct IO_Resource_s {
  IO_DES      IO_Header;
  IO_RANGE    IO_Data[ANYSIZE_ARRAY];
} IO_RESOURCE, *PIO_RESOURCE;
```

The IO_RESOURCE structure is used for specifying either a resource list or a resource requirements list that describes I/O port usage for a device instance.

Members

IO_Header

An IO_DES structure.

IO_Data

For a resource list:

Zero.

For a resource requirements list:

An IO_RANGE array.

IRQ_DES

```
typedef struct IRQ_Des_s {
  DWORD    IRQD_Count;
  DWORD    IRQD_Type;
  DWORD    IRQD_Flags;
  ULONG    IRQD_Alloc_Num;
  ULONG    IRQD_Affinity;
} IRQ_DES, *PIRQ_DES;
```

The IRQ_DES structure is used for specifying either a resource list or a resource requirements list that describes IRQ line usage for a device instance.

Members

IRQD_Count

For a resource list:

Zero.

For a resource requirements list:

The number of elements in the IRQ_RANGE array that is included in the IRQ_RESOURCE structure.

IRQD_Type

Must be set to the constant value **IRQType_Range**.

IRQD_Flags

One bit flag from *each* of the flag sets described in the following table.

	Flag	Definition
Sharing Flags		
	fIRQD_Exclusive	The IRQ line cannot be shared.
	fIRQD_Share	The IRQ line can be shared.
	mIRQD_Share	Bit mask for the bits within **IRQD_Flags** that specify the sharing value.

Flag	Definition
Triggering Flags	
fIRQD_Level	The IRQ line is level-triggered.
fIRQD_Edge	The IRQ line is edge-triggered.
mIRQD_Edge_Level	Bit mask for the bits within **IRQD_Flags** that specify the triggering value.

IRQD_Alloc_Num

For a resource list:

The number of the IRQ line that is allocated to the device.

For a resource requirements list:

Not used.

IRQD_Affinity

For a resource list:

A bit mask representing the processor affinity of the IRQ line that is allocated to the device. Bit zero represents the first processor, bit two the second, and so on. Set this value to -1 to represent all processors.

For a resource requirements list:

Not used.

IRQ_RANGE

```
typedef struct IRQ_Range_s {
  ULONG    IRQR_Min;      // minimum IRQ in the range
  ULONG    IRQR_Max;      // maximum IRQ in the range
  ULONG    IRQR_Flags;    // flags describing the range (fIRQD flags)
} IRQ_RANGE, *PIRQ_RANGE;
```

The IRQ_RANGE structure specifies a resource requirements list that describes IRQ line usage for a device instance.

Members

IRQR_Min

The lowest-numbered of a range of contiguous IRQ lines that can be allocated to the device.

IRQR_Max

The highest-numbered of a range of contiguous IRQ lines that can be allocated to the device.

IRQR_Flags

One bit flag from *each* of the flag sets described in the table included with the description of the **IRQD_Flags** member of the IRQ_DES structure.

IRQ_RESOURCE

```
typedef struct IRQ_Resource_s {
  IRQ_DES       IRQ_Header;
  IRQ_RANGE     IRQ_Data[ANYSIZE_ARRAY];
} IRQ_RESOURCE, *PIRQ_RESOURCE;
```

The IRQ_RESOURCE structure is used for specifying either a resource list or a resource requirements list that describes IRQ line usage for a device instance.

Members

IO_Header

An IRQ_DES structure.

IO_Data

For a resource list:

Zero.

For a resource requirements list:

An IRQ_RANGE array.

MEM_DES

```
typedef struct Mem_Des_s {
  DWORD         MD_Count;
  DWORD         MD_Type;
  DWORDLONG     MD_Alloc_Base;
  DWORDLONG     MD_Alloc_End;
  DWORD         MD_Flags;
  DWORD         MD_Reserved;
} MEM_DES, *PMEM_DES;
```

The MEM_DES structure is used for specifying either a resource list or a resource requirements list that describes memory usage for a device instance.

Members

MD_Count

For a resource list:

Zero.

For a resource requirements list:

The number of elements in the MEM_RANGE array that is included in the MEM_RESOURCE structure.

MD_Type

Must be set to the constant value **MType_Range**.

MD_Alloc_Base

For a resource list:

The lowest-numbered of a range of contiguous physical memory addresses allocated to the device.

For a resource requirements list:

Zero.

MD_Alloc_End

For a resource list:

The highest-numbered of a range of contiguous physical memory addresses allocated to the device.

For a resource requirements list:

Zero.

MD_Flags

One bit flag from *each* of the flag sets described in the following table.

Flag	Definition
Read-Only Flags	
fMD_ROM	The specified memory range is read-only.
fMD_RAM	The specified memory range is not read-only.
mMD_MemoryType	Bit mask for the bit within **MD_Flags** that specifies the read-only attribute.

Continued

Flag	Definition
Write-Only Flags	
fMD_ReadDisallowed	The specified memory range is write-only.
fMD_ReadAllowed	The specified memory range is not write-only.
mMD_Readable	Bit mask for the bit within **MD_Flags** that specifies the write-only attribute.
Address Size Flags	
fMD_24	24-bit addressing (*not used*).
fMD_32	32-bit addressing.
mMD_32_24	Bit mask for the bit within **MD_Flags** that specifies the address size.
Pre-Fetch Flags	
fMD_PrefetchAllowed	The specified memory range can be prefetched.
fMD_PrefetchDisallowed	The specified memory range cannot be prefetched.
mMD_Prefetchable	Bit mask for the bit within **MD_Flags** that specifies the prefetch ability.
Caching Flags	
fMD_Cacheable	The specified memory range can be cached.
fMD_NonCacheable	The specified memory range cannot be cached.
mMD_Cacheable	Bit mask for the bit within **MD_Flags** that specifies the caching ability.
Combined-Write Caching Flags	
fMD_CombinedWriteAllowed	Combined-write caching is allowed.
fMD_CombinedWriteDisallowed	Combined-write caching is not allowed.
mMD_CombinedWrite	Bit mask for the bit within **MD_Flags** that specifies the combine-write caching ability.

MD_Reserved

For internal use only.

MEM_RANGE

```
typedef struct Mem_Range_s {
  DWORDLONG    MR_Align;
  ULONG        MR_nBytes;
  DWORDLONG    MR_Min;
  DWORDLONG    MR_Max;
  DWORD        MR_Flags;
  DWORD        MR_Reserved;
} MEM_RANGE, *PMEM_RANGE;
```

The MEM_RANGE structure specifies a resource requirements list that describes memory usage for a device instance.

Members

MR_Align

Mask used to specify the memory address boundary on which the first allocated memory address must be aligned.

MR_nBytes

The number of bytes of memory required by the device.

MR_Min

The lowest-numbered of a range of contiguous memory addresses that can be allocated to the device.

MR_Max

The highest-numbered of a range of contiguous memory addresses that can be allocated to the device.

MR_Flags

One bit flag from *each* of the flag sets described in the table included with the description of the **MD_Flags** member of the MEM_DES structure.

MR_Reserved

For internal use only.

MEM_RESOURCE

```
 ypedef struct Mem_Resource_s {
  MEM_DES       MEM_Header;
  MEM_RANGE     MEM_Data[ANYSIZE_ARRAY];
} MEM_RESOURCE, *PMEM_RESOURCE;
```

The MEM_RESOURCE structure is used for specifying either a resource list or a resource requirements list that describes memory usage for a device instance.

Members

MEM_Header

A MEM_DES structure.

MEM_Data

For a resource list:

Zero.

For a resource requirements list:

A MEM_RANGE array.

MFCARD_DES

```
typedef struct MfCard_Des_s {
  DWORD     PMF_Count;
  DWORD     PMF_Type;
  DWORD     PMF_Flags;
  BYTE      PMF_ConfigOptions;
  BYTE      PMF_IoResourceIndex;
  BYTE      PMF_Reserved[2];
  DWORD     PMF_ConfigRegisterBase;
} MFCARD_DES, *PMFCARD_DES;
```

The MFCARD_DES structure is used for specifying either a resource list or a resource requirements list that describes resource usage by *one* of the hardware functions provided by an instance of a multifunction device.

Members

PMF_Count

Must be 1.

PMF_Type

Not used.

PMF_Flags

One bit flag is defined, as described in the following table.

Flag	Definition
fPMF_AUDIO_ENABLE	If set, audio is enabled.

PMF_ConfigOptions

Contents of the 8-bit PCMCIA Configuration Option Register.

PMF_IoResourceIndex

Zero-based index indicating the IO_RESOURCE structure that describes the I/O resources for the hardware function being described by this MFCARD_DES structure.

PMF_Reserved

Not used.

PMF_ConfigRegisterBase

Offset from the beginning of the card's attribute memory space to the base configuration register address.

MFCARD_RESOURCE

```
typedef struct MfCard_Resource_s {
  MFCARD_DES    MfCard_Header;
} MFCARD_RESOURCE, *PMFCARD_RESOURCE;
```

The MFCARD_RESOURCE structure is used for specifying either a resource list or a resource requirements list that describes resource usage by *one* of the hardware functions provided by an instance of a multifunction device.

Members

MfCard_Header

A MFCARD_DES structure.

PCCARD_DES

```
typedef struct PcCard_Des_s {
  DWORD    PCD_Count;
  DWORD    PCD_Type;
  DWORD    PCD_Flags;
  BYTE     PCD_ConfigIndex;
  BYTE     PCD_Reserved[3];
  DWORD    PCD_MemoryCardBase1;
  DWORD    PCD_MemoryCardBase2;
} PCCARD_DES, *PPCCARD_DES;
```

The PCCARD_DES structure is used for specifying either a resource list or a resource requirements list that describes resource usage by a PCMCIA card instance.

Members

PCD_Count

Must be 1.

PCD_Type

Not used.

PCD_Flags

One bit flag from *each* of the flag sets described in the following table.

Flag	Definition
I/O Addressing Flags	
fPCD_IO_8	The device uses 8-bit I/O addressing.
fPCD_IO_16	The device uses 16-bit I/O addressing.
mPCD_IO_8_16	Bit mask for the bit within **PCD_Flags** that specifies 8-bit or 16-bit I/O addressing.
Memory Addressing Flags	
fPCD_MEM_8	The device uses 8-bit memory addressing.
fPCD_MEM_16	The device uses 16-bit memory addressing.
mPCD_MEM_8_16	Bit mask for the bit within **PCD_Flags** that specifies 8-bit or 16-bit memory addressing.

PCD_ConfigIndex

The 8-bit index value used to locate the device's configuration.

PCD_Reserved[3]

Not used.

PCD_MemoryCardBase1

Optional, card base address of the first memory window.

PCD_MemoryCardBase2

Optional, card base address of the second memory window.

PCCARD_RESOURCE

```
typedef struct PcCard_Resource_s {
   PCCARD_DES    PcCard_Header;
} PCCARD_RESOURCE, *PPCCARD_RESOURCE;
```

The PCCARD_RESOURCE structure is used for specifying either a resource list or a resource requirements list that describes resource usage by a PCMCIA card instance.

Members

PcCard_Header

A PCCARD_DES structure.

Other Structures

This sections describes additional PnP Configuration Manager structures.

CONFLICT_DETAILS

```
typedef struct _CONFLICT_DETAILS_W {
  ULONG      CD_ulSize;
  ULONG      CD_ulMask;
  DEVINST    CD_dnDevInst;
  RES_DES    CD_rdResDes;
  ULONG      CD_ulFlags;
  WCHAR      CD_szDescription[MAX_PATH];
} CONFLICT_DETAILS_W , *PCONFLICT_DETAILS_W;
```

The CONFLICT_DETAILS structure is used as a parameter to the **CM_Get_Resource_ Conflict_Details** function.

Members

CD_ulSize

Size, in bytes, of the CONFLICT_DETAILS structure.

CD_ulMask

One or more bit flags supplied by the caller of **CM_Get_Resource_Conflict_Details**. The bit flags are described in the following table.

Flag	Description
CM_CDMASK_DEVINST	If set, **CM_Get_Resource_Conflict_Details** supplies a value for the **CD_dnDevInst** member.
CM_CDMASK_RESDES	*Not used.*
CM_CDMASK_FLAGS	If set, **CM_Get_Resource_Conflict_Details** supplies a value for the **CD_ulFlags** member.
CM_CDMASK_DESCRIPTION	If set, **CM_Get_Resource_Conflict_Details** supplies a value for the **CD_szDescription** member.

CD_dnDevInst

If CM_CDMASK_DEVINST is set in **CD_ulMask**, this member will receive a handle to a device instance that has conflicting resources. If a handle is not obtainable, the member receives -1.

CD_rdResDes

Not used.

CD_ulFlags

If CM_CDMASK_FLAGS is set in **CD_ulMask**, this member can receive bit flags listed in the following table.

Flag	Description
CM_CDFLAGS_DRIVER	If set, the string contained in the **CD_szDescription** member represents a driver name instead of a device name, and **CD_dnDevInst** is -1.
CM_CDFLAGS_ROOT_OWNED	If set, the conflicting resources are owned by the root device (that is, the HAL), and **CD_dnDevInst** is -1.
CM_CDFLAGS_RESERVED	If set, the owner of the conflicting resources cannot be determined, and **CD_dnDevInst** is -1.

CD_szDescription

If CM_CDMASK_DESCRIPTION is set in **CD_ulMask**, this member will receive a NULL-terminated text string representing a description of the device that owns the resources. If CM_CDFLAGS_DRIVER is set in **CD_ulFlags**, this string represents a driver name. If CM_CDFLAGS_ROOT_OWNED or CM_CDFLAGS_RESERVED is set, the string value is NULL.

PnP Configuration Manager Types

This section describes PnP Configuration Manager data types.

PNP_VETO_TYPE

```
typedef enum _PNP_VETO_TYPE {
  PNP_VetoTypeUnknown,
  PNP_VetoLegacyDevice,
  PNP_VetoPendingClose,
  PNP_VetoWindowsApp,
  PNP_VetoWindowsService,
  PNP_VetoOutstandingOpen,
  PNP_VetoDevice,
  PNP_VetoDriver,
  PNP_VetoIllegalDeviceRequest,
  PNP_VetoInsufficientPower,
  PNP_VetoNonDisableable,
  PNP_VetoLegacyDriver,
}   PNP_VETO_TYPE, *PPNP_VETO_TYPE;
```

If the PnP Manager rejects a request to perform an operation, the PNP_VETO_TYPE enumeration is used for identifying the reason for the rejection.

Enumerators

PNP_VetoTypeUnknown

The specified operation was rejected for an unknown reason.

PNP_VetoLegacyDevice

The device does not support the specified PnP operation.

PNP_VetoPendingClose

The specified operation cannot be completed because of a pending close operation.

PNP_VetoWindowsApp

A Microsoft® Win32® application vetoed the specified operation.

PNP_VetoWindowsService

A Win32 service vetoed the specified operation.

PNP_VetoOutstandingOpen

The requested operation was rejected because of outstanding open handles.

PNP_VetoDevice

The device supports the specified operation, but the device rejected the operation.

PNP_VetoDriver

The driver supports the specified operation, but the driver rejected the operation.

PNP_VetoIllegalDeviceRequest

The device does not support the specified operation.

PNP_VetoInsufficientPower

There is insufficient power to perform the requested operation.

PNP_VetoNonDisableable

The device cannot be disabled.

PNP_VetoLegacyDriver

The driver does not support the specified PnP operation.

C H A P T E R 8

Device Setup Classes

To facilitate device installation, devices that are set up and configured in the same way are grouped into a device setup class. For example, SCSI media changer devices are grouped into the MediumChanger device setup class. The device setup class defines such things as the class installer and class coinstallers that are involved in installing the device.

Microsoft defines setup classes for most devices. IHVs and OEMs can define new device setup classes, but only if none of the existing classes apply. For example, a camera vendor might think they need to define a new setup class, but cameras fall under the Image class. Similarly, UPS devices fall under the Battery class.

There is a GUID associated with each device setup class. System-defined setup class GUIDs are defined in *devguid.h* and typically have symbollic names of the form GUID_ DEVCLASS_*XXX*.

The device setup class GUID defines the **..\CurrentControlSet\Control\Class\***ClassGUID* registry key under which to create a new subkey for any particular device of a standard setup class.

This chapter lists the system-defined device setup classes. In the definition for a given class, the **Class** and **ClassGuid** entries correspond to the values that must be specified in the *INF Version Section* of a device's INF file.

Supplying the appropriate class GUID value in the INF for a device, rather than or in addition to the **Class=***class-name* entry, improves the performance of system INF searching significantly. In fact, system INFs that do not require either entry, such as those that neither install a new device class installer nor a device driver, sometimes supply **Class-Guid=**{00000000-0000-0000-0000-000000000000} in their **Version** sections to cut down on the system's INF searching time.

1394 Host Bus Controller

Class = 1394
ClassGuid = {6bdd1fc1-810f-11d0-bec7-08002be2092f}

This class includes system-supplied drivers of 1394 host controllers connected on a PCI bus, but not drivers of 1394 peripherals.

Battery Devices

Class = Battery
ClassGuid = {72631e54-78a4-11d0-bcf7-00aa00b7b32a}

This class includes drivers of battery devices and UPSes.

CD-ROM Drives

Class = CDROM
ClassGuid = {4d36e965-e325-11ce-bfc1-08002be10318}

This class includes drivers of CD-ROM drives, including SCSI CD-ROM drives. By default, the system's CD-ROM class installer also installs a system-supplied CD audio driver and CD-ROM changer driver as PnP filters.

Disk Drives

Class = DiskDrive
ClassGuid = {4d36e967-e325-11ce-bfc1-08002be10318}

This class includes drivers of hard disk drives. See also the HDC and SCSIAdapter classes.

Display Adapters

Class = Display
ClassGuid = {4d36e968-e325-11ce-bfc1-08002be10318}

This class includes drivers of video adapters, including display drivers and video miniports.

Floppy Disk Controllers

Class = FDC
ClassGuid = {4d36e969-e325-11ce-bfc1-08002be10318}

This class includes drivers of floppy disk drive controllers.

Floppy Disk Drives

Class= FloppyDisk
ClassGuid= {4d36e980-e325-11ce-bfc1-08002be10318}

This class includes drivers of floppy drives.

Hard Disk Controllers

Class = HDC
ClassGuid = {4d36e96a-e325-11ce-bfc1-08002be10318}

This class includes drivers of hard disk controllers, including ATA/ATAPI controllers but not SCSI and RAID disk controllers.

Human Input Devices (HID)

Class = HIDClass
ClassGuid = {745a17a0-74d3-11d0-b6fe-00a0c90f57da}

This class includes devices that export interfaces of the HID class, including HID keyboard and mouse devices, which the installed HID device drivers enumerate as their respective "child" devices. (See also the Keyboard or Mouse classes later in this list.)

Imaging Device

Class = Image
ClassGuid = {6bdd1fc6-810f-11d0-bec7-08002be2092f}

This class includes drivers of still-image capture devices, digital cameras, and scanners.

IrDA Devices

Class = Infrared
ClassGuid = {6bdd1fc5-810f-11d0-bec7-08002be2092f}

This class includes Serial-IR and Fast-IR NDIS miniports, but see also the Network Adapter class for other NDIS NIC miniports.

Keyboard

Class = Keyboard
ClassGuid = {4d36e96b-e325-11ce-bfc1-08002be10318}

This class includes all keyboards. That is, it also must be specified in the (secondary) INF for an enumerated "child" HID keyboard device.

Medium Changers

Class= MediumChanger
ClassGuid= {ce5939ae-ebde-11d0-b181-0000f8753ec4}

This class includes drivers of SCSI media changer devices.

Memory Technology Driver

Class = MTD
ClassGUID = {4d36e970-e325-11ce-bfc1-08002be10318}

This class includes drivers for memory devices, such as flash memory cards.

Multimedia

Class = Media
ClassGuid = {4d36e96c-e325-11ce-bfc1-08002be10318}

This class includes Audio and DVD multimedia devices, joystick ports, and full-motion video-capture devices.

Modem

Class = Modem
ClassGuid = {4d36e96d-e325-11ce-bfc1-08002be10318}

This class installs modems. An INF for a device of this class installs no device driver(s), but rather specifies the features and configuration information of a particular modem and stores this information in the registry. See also the Multifunction class.

Monitor

Class = Monitor
ClassGuid = {4d36e96e-e325-11ce-bfc1-08002be10318}

This class includes display monitors. An INF for a device of this class installs no device driver(s), but rather specifies the features of a particular monitor to be stored in the registry for use by drivers of video adapters. (Monitors are enumerated as the child devices of display adapters.)

Mouse

Class = Mouse
ClassGuid = {4d36e96f-e325-11ce-bfc1-08002be10318}

This class includes all mice and other kinds of pointing devices, such as trackballs. That is, it also must be specified in the (secondary) INF for an enumerated "child" HID mouse device.

Multifunction Devices

Class = Multifunction
ClassGuid = {4d36e971-e325-11ce-bfc1-08002be10318}

This class includes combo cards, such as a PCMCIA modem and netcard adapter. The driver for such a PnP multifunction device is installed under this class and enumerates the modem and netcard separately as its "child" devices.

Multi-port Serial Adapters

Class = MultiportSerial
ClassGuid = {50906cb8-ba12-11d1-bf5d-0000f805f530}

This class includes intelligent multiport serial cards, but not peripheral devices that connect to its ports. It does not include unintelligent (16550-type) mutiport serial controllers or single-port serial controllers (see the Ports class).

Network Adapter

Class = Net
ClassGuid = {4d36e972-e325-11ce-bfc1-08002be10318}

This class includes NDIS NIC miniports excluding Fast-IR miniports, NDIS intermediate drivers (of "virtual adapters"), and CoNDIS MCM miniports.

Network Client

Class = NetClient
ClassGuid = {4d36e973-e325-11ce-bfc1-08002be10318}

This class includes network and/or print providers.

Network Service

Class = NetService
ClassGuid = {4d36e974-e325-11ce-bfc1-08002be10318}

This class includes network services, such as redirectors and servers.

Network Transport

Class = NetTrans
ClassGuid = {4d36e975-e325-11ce-bfc1-08002be10318}

This class includes NDIS protocols, CoNDIS stand-alone call managers, and CoNDIS clients, as well as higher level drivers in transport stacks.

PCMCIA Adapters

Class = PCMCIA
ClassGuid = {4d36e977-e325-11ce-bfc1-08002be10318}

This class includes system-supplied drivers of PCMCIA and CardBus host controllers, but not drivers of PCMCIA or CardBus peripherals.

Ports (COM & LPT serial ports)

Class = Ports
ClassGuid = {4d36e978-e325-11ce-bfc1-08002be10318}

This class includes drivers of serial or parallel port devices, but see also the MultiportSerial class.

Printer

Class = Printer
ClassGuid = {4d36e979-e325-11ce-bfc1-08002be10318}

This class includes printers.

SCSI and RAID Controllers

Class = SCSIAdapter
ClassGuid = {4d36e97b-e325-11ce-bfc1-08002be10318}

This class includes SCSI HBA miniports and disk-array controller drivers.

Smart Card Readers

Class = SmartCardReader
ClassGuid = {50dd5230-ba8a-11d1-bf5d-0000f805f530}

This class includes drivers for smart card readers.

Storage Volumes

Class = Volume
ClassGuid = {71a27cdd-812a-11d0-bec7-08002be2092f}

This class includes storage volumes as defined by the system-supplied logical volume manager and class drivers that create device objects to represent storage volumes, such as the system disk class driver.

System Devices

Class = System
ClassGuid = {4d36e97d-e325-11ce-bfc1-08002be10318}

This class includes the Windows® 2000 HALs, system bus drivers, the system ACPI driver, and the system volume-manager driver. It also includes battery drivers and UPS drivers.

Tape Drives

Class = TapeDrive
ClassGuid = {6d807884-7d21-11cf-801c-08002be10318}

This class includes drivers of tape drives, including all tape miniclass drivers.

USB

Class = USB
ClassGuid = {36fc9e60-c465-11cf-8056-444553540000}

This class includes system-supplied (bus) drivers of USB host controllers and drivers of USB hubs, but not drivers of USB peripherals.

The following classes and GUIDs should not be used to install devices (or drivers) on Windows 2000 platforms:

Adapter

Class = Adapter
ClassGUID = {4d36e964-e325-11ce-bfc1-08002be10318}

This class is obsolete.

APM

Class = APMSupport
ClassGUID = {d45b1c18-c8fa-11d1-9f77-0000f805f530}

This class is reserved for system use.

Computer

Class = Computer
ClassGUID = {4d36e966-e325-11ce-bfc1-08002be10318}

This class is reserved for system use.

Decoders

Class = Decoder
ClassGUID = {6bdd1fc2-810f-11d0-bec7-08002be2092f}

This class is reserved for future use.

Global Positioning System

Class = GPS
ClassGUID = {6bdd1fc3-810f-11d0-bec7-08002be2092f}

This class is reserved for future use.

No driver

Class = NoDriver
ClassGUID = {4d36e976-e325-11ce-bfc1-08002be10318}

This class is obsolete.

Non-Plug and Play Drivers

Class = LegacyDriver
ClassGUID = {8ecc055d-047f-11d1-a537-0000f8753ed1}

This class is reserved for system use.

Other Devices

Class = Unknown
ClassGUID = {4d36e97e-e325-11ce-bfc1-08002be10318}

This class is reserved for system use. Enumerated devices for which the system cannot determine the type are installed under this class. Do not use this class if you're unsure in which class your device belongs; either determine the correct device setup class or create a new class.

Printer Upgrade

Class = Printer Upgrade
ClassGUID = {4d36e97a-e325-11ce-bfc1-08002be10318}

This class is reserved for system use.

Sound

Class = Sound
ClassGUID = {4d36e97c-e325-11ce-bfc1-08002be10318}

This class is obsolete.

CHAPTER 9

The txtsetup.oem File Format

During the text-mode setup phase of Windows® 2000 installation, the Setup program installs drivers for devices that are required to boot the machine. Most of these drivers are included with the operating system. A vendor can enable users to install an additional driver during text-mode setup by supplying a *txtsetup.oem* file on a floppy disk. This chapter describes the format of a *txtsetup.oem* file. See the *Plug and Play, Power Management, and Setup Design Guide* for an overview of text-mode setup installation and a general discussion of installing a device required to boot the machine.

A *txtsetup.oem* file consists of several sections that use the following general format:

```
[SectionName]
entry = value1,value2,...
```

The name of the section is enclosed in square brackets ([ ]). A pound sign (#) or semicolon character (;) at the beginning of a line indicates a comment. Strings with embedded spaces, commas, or hashes must be enclosed in double quotes (" ").

A *txtsetup.oem* file must include the following sections:

- A **Disks** section

  See *Disks Section of a txtsetup.oem File* for further information.

- A **Defaults** section

  See *Defaults Section of a txtsetup.oem File* for further information.

- One or more *HwComponent* sections

  See *HwComponent Section of a txtsetup.oem File* for further information.

- One or more **Files.***HwComponent.ID* sections

  See *Files.HwComponent.ID Section of a txtsetup.oem File* for further information.

- One or more **Config.***DriverKey* sections

  See *Config.DriverKey Section of a txtsetup.oem File* for further information.

A *txtsetup.oem* file for a PnP mass storage device must also include the following section:

- A **HardwareIds.scsi.***Service* section

  See *HardwareIds.scsi.Service Section of a txtsetup.oem File* for further information.

Disks Section of a txtsetup.oem File

The **Disks** section identifies the disks in the device installation kit. This section has the following format:

[Disks]
diskN = "description",tagfile,directory
...

diskN

Specifies a key that can be used in subsequent sections to identify the disk.

description

Specifies a string containing the name of the disk. Setup uses the description to prompt the user to insert the disk.

tagfile

Specifies the name of a verification file on the disk. The filename must be specified as a full path from the root and must not specify a drive. Setup checks for this file to ensure that the user inserted the correct disk.

directory

Specifies the directory on the disk where the installation files are located. The directory must be specified as a full path from the root and must not specify a drive.

The following example shows a **Disks** section for an installation kit with two disks:

```
[Disks]
disk1 = "OEM SCSI driver disk 1",\disk1.tag,\
disk2 = "OEM SCSI driver disk 2",\disk2.tag,\
; ...
```

Defaults Section of a txtsetup.oem File

The **Defaults** section lists the default driver(s) for each hardware component supported by this file. Setup highlights the default selection when it presents a list of drivers to the user.

[Defaults]
component = *ID*
...

component

Specifies a hardware component supported by this file. The *component* must be one of the following system-defined values: **computer** or **scsi**.

ID

Specifies a string that identifies the default option. This string matches an ID specified in the corresponding *HwComponent* section.

If a *txtsetup.oem* file fails to define a default driver for a supported component, Setup uses the first entry in the *HwComponent* section.

The following example shows a **Defaults** section (and the *HwComponent* section) for a *txtsetup.oem* file that supports one component (**scsi**):

```
; ...
[Defaults]
SCSI = oemscsi

[SCSI]                  ; HwComponent section
oemscsi = "OEM Fast SCSI Controller"
oemscsi2 = "OEM Fast SCSI Controller 2"

; ...
```

HwComponent Section of a txtsetup.oem File

A *HwComponent* section lists the drivers available for a particular component. There is a *HwComponent* section for each type of component supported by the file.

[*HwComponent*]
ID = *description*
...

HwComponent

The name of the section must be one of the following system-defined values: **computer** or **scsi**.

ID

Specifies a string, unique within this section, that identifies the option.

For each entry in this section, there must be a corresponding **Files.***HwComponent.ID* section in the file.

For the **computer** component, the last three characters of the string determine which kernel Setup copies. If this string ends in "_up", Setup copies the uniprocessor kernel. If this string ends in "_mp", Setup copies the multiprocessor kernel. If the string does not end in "_Xp", Setup copies one or the other kernel, but does not guarantee which one.

description

Specifies a string that Setup presents to the user in the menu of driver choices.

The following example shows a *HwComponent* section for a *txtsetup.oem* file that supports one component (**scsi**) and offers two options:

```
; ...
[SCSI]                  ; HwComponent section
oemscsi = "OEM Fast SCSI Controller"
oemscsi2 = "OEM Fast SCSI Controller 2"

; ...
```

Files.HwComponent.ID Section of a txtsetup.oem File

A **Files.***HwComponent.ID* section lists the files to be copied if the user selects a particular component option. One of these sections must be present for each option listed in each *HwComponent* section.

[Files.*HwComponent.ID***]**
filetype = *diskN*,*filename*[,*DriverKey*]
...

Files.HwComponent.ID

HwComponent corresponds to the name of a *HwComponent* section in the file. *ID* corresponds to an *ID* entry in that *HwComponent* section.

filetype

Identifies the type of the file to be copied. One of these entries is present for each file to be copied for this *HwComponent.ID*.

The *filetype* is one of the following system-defined values:

driver

Valid for all components. Setup copies the file to *%systemroot%\system32\drivers.*

dll

Valid for all components. Useful for the GDI portion of a display driver. Setup copies the file to *%systemroot%\system32.*

hal

Valid only for the **computer** component. Setup copies the file to *%systemroot%\system32\hal.dll* (for x86) or to *\os\winnt\hal.dll* on the system partition (for non-x86).

inf

Valid for all components. Specifies the regular INF file for the device. This file is used during GUI-mode setup and for other device maintenance operations. The file is copied to *%systemroot%\system32*.

catalog

Valid for drivers. Specifies a catalog file for the device. Not required for any component. For example, **catalog** = d1, *mydriver.cat*. See the WHQL guidelines for more information on catalog files.

detect

Valid for the **computer** component (x86 only). If specified, replaces the standard x86 hardware recognizer. Setup copies the file to *c:\ntdetect.com*.

diskN

Identifies the disk from which to copy the file. This value must match an entry in the **Disks** section.

filename

Specifies the name of the file, not including the directory path or drive. To form the full file name, Setup appends the *filename* to the directory specified for the disk in the **Disks** section.

DriverKey

Specifies the name of the key to be created in the registry services tree for this file, if the file is of type **driver**. This value is used to form **Config.***DriverKey* section names. This value is required for components of type **scsi**.

The following example shows a **Files.***HwComponent.ID* section in a *txtsetup.oem* file:

```
; ...
[Files.SCSI.oemscsi]
driver = d1,oemfs2.sys,OEMSCSI
inf = d1,oemsetup.inf
dll = d1, oemdrv.dll
catalog = d1, oemdrv.cat
; ...
```

Config.DriverKey Section of a txtsetup.oem File

A **Config.***DriverKey* section specifies values to be set in the registry for particular component options. Setup automatically creates the required values in the **Services\***DriverKey* key. Use this section to specify additional keys to be created under **Services\***DriverKey* and values under **Services\***DriverKey* and **Services\***DriverKey\subkey_name*.

[**Config.***DriverKey*]
value = *subkey_name,value_name,value_type,value*
...

subkey_name

Specifies the name of a key under the **Services\***DriverKey* tree where Setup places the specified value. Setup creates the key if it does not exist.

If *subkey_name* is the empty string (""), the value is placed under the **Services\***DriverKey*.

The *subkey_name* can specify more than one level of subkey, such as "subkey1\subkey2\subkey3".

value_name

Specifies the name of the value to be set.

value_type

Specifes the type of the registry entry. The *value_type* can be one of the following:

REG_DWORD

One *value* is allowed; it must be a string of 1-8 hex digits.

For example:

```
value = parameters,NumberOfButtons,REG_DWORD,0X2
```

REG_SZ or REG_EXPAND_SZ

One *value* is allowed; it is interpreted as the zero-terminated string to be stored.

For example:

```
value = parameters,Description,REG_SZ,"This is a text string"
```

REG_BINARY

One *value* is allowed; it is a string of hex digits, each pair of which is interpreted as a byte value.

For example (stores the byte stream 00,34,ec,4d,04,5a):

```
value = parameters,Data,REG_BINARY,0034eC4D045a
```

REG_MULTI_SZ

Multiple *value* arguments are allowed; each is interpreted as a component of the MULTI_SZ string.

For example:

```
value = parameters,Strings,REG_MULTI_SZ,String1,"String 2",string3
```

value

Specifies the value; its format depends on *value_type*.

The following example shows a **Config.***DriverKey* section:

```
; ...
[Config.OEMSCSI]
value = "",tag,REG_DWORD,5
value = parameters\PnpInterface,5,REG_DWORD,1
; ...
```

HardwareIds.scsi.Service Section of a txtsetup.oem File

A **HardwareIds.scsi.***Service* section specifies the hardware IDs of the devices that a particular mass-storage driver supports.

[HardwareIds.scsi.*Service***]**
id = "*deviceID*","*service*"
...

HardwareIds.scsi.*Service*

Service specifies the service to be installed.

deviceId

Specifies the device ID for a mass-storage device.

service

Specifies the service to be installed for the device.

The following example excerpt shows a ***HardwareIds.scsi.**Service* section for a disk device:

```
; ...
[HardwareIds.scsi.oemscsi]
id = "PCI\VEN_9004&DEV_8111","oemscsi"

; ...
```

- **Support Services.** Microsoft may, but is not obligated to, provide you with support services related to the SOFTWARE PRODUCT ("Support Services"). Use of Support Services is governed by the Microsoft policies and programs described in the user manual, in "online" documentation, and/or in other Microsoft-provided materials. Any supplemental software code provided to you as part of the Support Services shall be considered part of the SOFTWARE PRODUCT and subject to the terms and conditions of this EULA. With respect to technical information you provide to Microsoft as part of the Support Services, Microsoft may use such information for its business purposes, including for product support and development. Microsoft will not utilize such technical information in a form that personally identifies you.
- **Software Transfer.** You may permanently transfer all of your rights under this EULA, provided you retain no copies, you transfer all of the SOFTWARE PRODUCT (including all component parts, the media and printed materials, any upgrades, this EULA, and, if applicable, the Certificate of Authenticity), **and** the recipient agrees to the terms of this EULA.
- **Termination.** Without prejudice to any other rights, Microsoft may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of the SOFTWARE PRODUCT and all of its component parts.

3. **COPYRIGHT.** All title and copyrights in and to the SOFTWARE PRODUCT (including but not limited to any images, photographs, animations, video, audio, music, text, SAMPLE CODE, REDISTRIBUTABLES, and "applets" incorporated into the SOFTWARE PRODUCT) and any copies of the SOFTWARE PRODUCT are owned by Microsoft or its suppliers. The SOFTWARE PRODUCT is protected by copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE PRODUCT like any other copyrighted material **except** that you may install the SOFTWARE PRODUCT on a single computer provided you keep the original solely for backup or archival purposes. You may not copy the printed materials accompanying the SOFTWARE PRODUCT.

4. **U.S. GOVERNMENT RESTRICTED RIGHTS.** The SOFTWARE PRODUCT and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software—Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is Microsoft Corporation/One Microsoft Way/Redmond, WA 98052-6399.

5. **EXPORT RESTRICTIONS.** You agree that you will not export or re-export the SOFTWARE PRODUCT, any part thereof, or any process or service that is the direct product of the SOFTWARE PRODUCT (the foregoing collectively referred to as the "Restricted Components"), to any country, person, entity, or end user subject to U.S. export restrictions. You specifically agree not to export or re-export any of the Restricted Components (i) to any country to which the U.S. has embargoed or restricted the export of goods or services, which currently include, but are not necessarily limited to, Cuba, Iran, Iraq, Libya, North Korea, Sudan, and Syria, or to any national of any such country, wherever located, who intends to transmit or transport the Restricted Components back to such country; (ii) to any end user who you know or have reason to know will utilize the Restricted Components in the design, development, or production of nuclear, chemical, or biological weapons; or (iii) to any end user who has been prohibited from participating in U.S. export transactions by any federal agency of the U.S. government. You warrant and represent that neither the BXA nor any other U.S. federal agency has suspended, revoked, or denied your export privileges.

DISCLAIMER OF WARRANTY

NO WARRANTIES OR CONDITIONS. MICROSOFT EXPRESSLY DISCLAIMS ANY WARRANTY OR CONDITION FOR THE SOFTWARE PRODUCT. THE SOFTWARE PRODUCT AND ANY RELATED DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OR CONDITION OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE PRODUCT REMAINS WITH YOU.

LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, MICROSOFT'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS EULA SHALL BE LIMITED TO THE GREATER OF THE AMOUNT ACTUALLY PAID BY YOU FOR THE SOFTWARE PRODUCT OR US$5.00; PROVIDED, HOWEVER, IF YOU HAVE ENTERED INTO A MICROSOFT SUPPORT SERVICES AGREEMENT, MICROSOFT'S ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT. BECAUSE SOME STATES AND JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

MISCELLANEOUS

This EULA is governed by the laws of the State of Washington USA, except and only to the extent that applicable law mandates governing law of a different jurisdiction.

Should you have any questions concerning this EULA, or if you desire to contact Microsoft for any reason, please contact the Microsoft subsidiary serving your country, or write: Microsoft Sales Information Center/One Microsoft Way/Redmond, WA 98052-6399.

Proof of Purchase

0-7356-0929-2

Do not send this card with your registration.
Use this card as proof of purchase if participating in a promotion or rebate offer on *Microsoft® Windows® 2000 Driver Development Kit*. Card must be used in conjunction with other proof(s) of payment such as your dated sales receipt—see offer details.

Microsoft® Windows® 2000 Driver Development Kit

WHERE DID YOU PURCHASE THIS PRODUCT?
CUSTOMER NAME

Microsoft®

mspress.microsoft.com

Microsoft Press, PO Box 97017, Redmond, WA 98073-9830

OWNER REGISTRATION CARD **Register Today!** 0-7356-0929-2

Return the bottom portion of this card to register today.

Microsoft® Windows® 2000 Driver Development Kit

FIRST NAME	MIDDLE INITIAL	LAST NAME
INSTITUTION OR COMPANY NAME		
ADDRESS		
CITY	STATE	ZIP
E-MAIL ADDRESS	( ) PHONE NUMBER	

U.S. and Canada addresses only. Fill in information above and mail postage-free.
Please mail only the bottom half of this page.

Microsoft Windows 2000

Driver Development Reference Volume 1

The essential reference to Plug and Play, power-management, setup, and 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 Plug and Play, power-management, and setup driver support in Windows 2000.

mspress.microsoft.com

Part No. 097-0002733

Microsoft®